COMPUTATIONAL INVESTIGATIONS OF MAXIMUM FLOW ALGORITHMS

A Thesis Submitted
in Partial Fulfilment of the Requirements
for the Degree of

Master of Technology

by

AJAY KUMAR MISHRA

to the

DEPARTMENT OF INDUSTRIAL & MANAGEMENT ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY, KANPUR
MAY, 1993

14/5/93

[illegible]

CERTIFICATE

his is to certify that the present work on "**Computational Investigations of Maximum low Algorithms**" by Ajay Kumar Mishra has been carried out under my supervision and as not been submitted elsewhere for the award of a degree.

[illegible]

(R. K. AHUJA)
Associate Professor
Industrial & Management Engineering
Indian Institute of Technology
Kanpur

May, 1993

ABSTRACT

The maximum flow problem has attracted the attention of researchers for decades and several algorithms have been developed successively which improve the worst-case running time of maximum flow algorithms. Not all of these algorithms improve the running times in practice. Several studies in the past have tested the empirical performance of the maximum flow algorithms. Most of these studies have been limited to a comparison of the CPU times taken by the implementations of these algorithms. Ahuja and Orlin [1993] have proposed a methodology to test the computational behavior of algorithms using representative operation counts. Using this methodology, our study investigates the computational behavior of ten maximum flow algorithms which are likely to be efficient in practice. They include the classical Dinic's and Karzanov's algorithms and recently developed shortest augmenting path algorithm, capacity scaling algorithm, several versions of the preflow-push algorithm and excess scaling algorithm. We test these algorithms on two types of networks: random layered network and random grid network for problem sizes as big as 10,000 nodes and 100,000 arcs. Using numerous illustrations, we develop insight into the empirical behavior of these algorithms. We find the bottleneck operations of these algorithms for small size networks as well as for the asymptotic case. We estimate the growth rate of the bottleneck operations as a function of the network size. We find a linear estimate of the CPU time taken by the algorithms as a function of certain representative operations performed by the algorithm. We also report insightful comparisons of these algorithms[1].

[1] This study is the culmination of the computational study of maximum flow algorithms started by Dr. R. K. Ahuja, Dr. J. B. Orlin and Dr. M. Kodialam in 1987 at MIT, Cambridge. The codes for the algorithms and the network generators used were developed by them and some preliminary testing was also done. This author has used and modified the codes, when necessary, and performed a full-scale testing of these algorithms.

Dedicated to

Anu,
Neeraj,
Siva, and
Vivek.

Their company was another benign influence at IIT/K.

ACKNOWLEDGEMENTS

I gratefully acknowledge the contribution of the following towards this thesis:

Dr. R. K. Ahuja guided my efforts. Had not "beyond words" become a cliche, I would prefer it to express my gratitude towards him. It was fun to work with him on a project started by him about five years ago. I learnt a great deal through our interaction. His extreme accessibility was a source of comfort as well as pressure for me.

TVLN Siva Kumar willingly served as a ready reference UNIX manual and an efficient debugger. He offered free but useful suggestions on several occasions.

Dr. S. Sadagopan, Dr. T. P. Bagchi, and Dr. K. S. Singh permitted a liberal use of the computing resources and stationery.

Mr. K. Mohan provided invaluable help in completing the write-up in time. Mr. Prakash Mishra helped in taking the final print out.

Prof. J. B. Orlin and Dr. M. Kodialam, along with Dr. Ahuja, worked on this project earlier. My thanks to them.

My class-mates made me feel diligent while working on this thesis.

I take this opportunity to express my appreciation for some persons who made my stay at IIT/K fruitful and memorable.

Dr. R. K. Ahuja inspired me as a person, researcher, and teacher.

The faculty of IME Dept. furthered my interest in academics and encouraged me in my endeavours.

It was a delight to enjoy the hospitality of Mrs. Ahuja, Mrs. Sadagopan, and Mrs. Kripa Shanker on several occasions.

I was fortunate to have persons like Anu, Siva, Neeraj, Sreekanth, Sankar and Vivek as my friends during my stay here. Sreekanth was my window to the world. My class-mates, research engineers and trainees added fizz to the life here.

I am grateful to the staff at IME Dept. for having a cooperative attitude towards me.

CONTENTS

1 INTRODUCTION, 1

1.1 Introduction, 1

1.2 Motivations, 2

1.3 Background of this study, 3

1.4 Highlights of main results, 4

1.5 Notation and Definitions, 5

1.6 Overview of the thesis, 6

2 LITERATURE SURVEY AND PRELIMINARIES, 8

2.1 Introduction, 8

2.2 Historical background, 8

2.3 Data Structures, 13

2.4 Network Generators, 17

2.5 Methodology for computational investigation, 20

2.6 Environment used for testing, 24

3 AUGMENTING PATH ALGORITHMS, 25

3.1 Introduction, 25

3.2 The shortest augmenting path algorithm, 25

3.3 Dinic's algorithm, 30

3.4 Capacity scaling algorithm, 32

3.5 Computational results for shortest augmenting path algorithms, 34

3.6 Computational results for Dinic's algorithm, 42

3.7 Computational results for capacity scaling algorithm, 47

4 PREFLOW-PUSH ALGORITHMS, 56

4.1 Introduction, 56

4.2 Karzanov's algorithm, 56

4.3 Preflow-push algorithms, 57

4.3(a) Highest Label version, 60

4.3(b) FIFO version, 62

4.3(c) Wave version, 62

4.3(d) Lowest level version, 63

4.4 Computational results for Karzanov's algorithm, 63

4.5 Computational results for preflow-push algorithms, 68

4.5(a) FIFO version, 68

4.5(b) Highest label version, 79

4.5(c) Wave version, 86

4.6 A comparison of some preflow-push algorithms, 86

5 EXCESS SCALING ALGORITHMS, 92

5.1 Introduction, 92

5.2 Excess scaling algorithm, 92

5.3 Stack scaling algorithm, 94

5.4 Wave scaling algorithm, 94

5.5 Computational results for excess scaling algorithm, 95

5.6 Computational results for stack scaling algorithm, 98

5.7 Computational results for wave scaling algorithm, 104

5.8 Comparison of scaling algorithms, 109

5.9 Comparison of stack scaling algorithm with efficient preflow-push algorithms, 114

6 CONCLUSIONS, 116

6.1 Conclusions, 116

6.2 Scope for future work, 119

REFERENCES, 121

CHAPTER 1

INTRODUCTION

1.1 INTRODUCTION

One of the most fundamental problems in combinatorial optimization is to send the maximum possible flow between two specified nodes in a capacitated network without exceeding the arc capacities. This problem is refered to as the maximum flow problem. Its intuitive appeal, mathematical simplicity, and wide applicability has made it a popular research topic among mathematicians, operations researchers and computer scientists.

The maximum flow problem arises in a wide variety of situations and in several forms. They occur directly in problems as diverse as the machine scheduling problem, flow of petroleum products in a pipeline network, assignment of computer modules to computer processors, optimal destruction of military targets and statistical security of data. Ahuja, Magnanti and Orlin [1993] describe these and several other applications of the maximum flow problem. The maximum flow problem also occurs as a subproblem in solving more complex problems such as the minimum cost flow problem, generalized flow problem and the parametric maximum flow problem. The maximum flow problems also arise while developing several results in combinatorics like those relating to network connectivity, matchings and covers in bipartite networks.

The maximum flow problem is distinguished by the long line of successive contributions researchers have made in improving the worst-case complexity of algorithms. Some, but not all, of these improvements have produced improvements in practice. Ahuja, Magnanti and Orlin [1991] provide a detailed survey of the maximum flow algorithms. In this thesis, we study the computational behavior of some maximum flow algorithms. We denote by n, the number of nodes; by m, the number of arcs; and by U, the largest arc capacity in the network. We have limited our study to the best previous maximum flow algorithms and some recent algorithms that are likely to be efficient in practice. We tested the following algorithms:

Augmenting path algorithms

(1) The $O(n^2m)$ augmenting path algorithm due to Dinic [1970].

(2) The $O(n^2m)$ distance directed augmenting path algorithm due to Ahuja and Orlin [1991].

(3) The $O(nm \log U)$ capacity scaling algorithm due to Gabow [1985].

Preflow-push algorithms

(4) The $O(n^3)$ preflow-push algorithm due to Karzanov [1974].

(5) The following versions of the preflow-push algorithms due to Goldberg and Tarjan [1986]:

(5a) The $O(n^2 \sqrt{m})$ highest label preflow-push algorithm.

(5b) The $O(n^3)$ first-in first-out (FIFO) preflow-push algorithm.

(5c) The $O(n^3)$ wave implementation of the preflow-push algorithm.

Excess scaling algorithms

(6) The following versions of the excess scaling algorithms due to Ahuja and Orlin [1989], and Ahuja, Orlin, and Tarjan [1989]:

(6a) The $O(nm + n^2 \log U)$ excess scaling algorithm.

(6b) The $O(nm + \frac{n^2 \log U}{\log \log U})$ stack scaling algorithm.

(6c) The $O(nm + n^2 \sqrt{\log U})$ wave scaling algorithm.

The worst-case complexities of algorithms mentioned in (5) and (6) above can be improved by using the dynamic tree data structure due to Sleator and Tarjan [1983]. However, we have not tested these implementations, because we believe that these implementations will not be competitive with the algorithms we tested due to the substantial overheads associated with the dynamic trees data structures. We also did not test the primal simplex algorithm for the maximum flow problem since we believe that this algorithm would run much slower than most of the algorithms we tested.

1.2 MOTIVATIONS

This study was primarily motivated by the desire to gain insight into the empirical behavior of maximum flow algorithms. Our secondary objective was to perform a comparative study of the maximum flow algorithms. Several computational studies have been reported in the literature but most of these have been limited to comparing the CPU times taken by specific implementations of the maximum flow algorithms. Also, few studies in the past have tested such an extensive range of classical, recent and some as yet untested maximum flow algorithms in similar conditions, for both small and very large size networks. We bring the highest distance version of preflow-push algorithms and excess scaling algorithms, computational testing on which has not yet been reported, together with the shortest augmenting path algorithm, capacity scaling algorithm, and the classical Dinic's and Karzanov's algorithms. Specifically, we have attempted to achieve the following objectives in this study.

(i) analysis of empirical running times as a function of the problem size parameters;

(ii) intuitive/rigorous explanations of the running times;

(iii) identification of the bottleneck operation(s);

(iv) study the effects of various time-saving heuristics;

(v) comparison of different algorithms.

1.3 BACKGROUND OF THIS STUDY

This computational study of maximum flow algorithms was started by Dr. R. K. Ahuja and Dr. J. B. Orlin at MIT, Cambridge in 1987. The codes for the maximum flow algorithms tested in this thesis and the network generators used for the testing were developed by them. The codes for the Dinic's and Karzanov's algorithms are those used by Imai [1983]. A comparison of the CPU time taken by the codes for the algorithms being tested was the widely used technique to test algorithms then. They put the testing in abeyance for want of a better testing methodology. After consultations with fellow researchers and statisticians and considerable research, in which Muralidharan Kodialam, a PhD student at OR Center, MIT, joined them, they developed a methodology for testing algorithms using representative operation counts. This is described in Ahuja and Orlin [1993] and Ahuja, Magnanti and Orlin [1993]. We use this methodology to conduct the computational investigations in our study and provide a brief description of their methodology in Section 2.5. This author joined the above-mentioned team to complete the study by using the codes available for the maximum flow algorithms and the network generators according to the above mentioned methodology.

Specifically, this author's contribution to this study is in using the existing codes and modifying them when necessary, to obtain, under the active guidance of his thesis supervisor, all the results reported in this thesis. Exhaustive experiments to choose the test problems were conducted during this thesis.

1.4 HIGHLIGHTS OF MAIN RESULTS

Some important results obtained during this study are:

(i) The shortest augmenting path algorithm and Dinic's algorithm have comparable empirical running times.

(ii) As the augmenting path algorithms proceed, each successive augmentation becomes more and more expensive.

(iii) Incorporating scaling worsens the running time of the augmenting path algorithms.

(iv) The preflow-push algorithms are substantially faster than the augmenting path algorithms.

(v) The highest label version of the preflow-push algorithm is faster than all the other algorithms we tested.

(vi) The excess scaling algorithms do not improve the empirical running time of the preflow-push algorithms.

(vi) The stack scaling version of the excess scaling algorithm has better empirical performance than other excess scaling algorithms.

(vii) The node relabeling time (or, the time to construct layered networks) is the asymptotic bottleneck operation for all the algorithms we tested.

(viii) We can obtain a good estimate of the CPU time taken by an algorithm by a function of some of its operations, called representative operations.

(ix) The methodology suggested by Ahuja and Orlin [1993] for testing algorithms, aids in developing good insight into an algorithm's empirical behavior.

1.5 NOTATION AND DEFINITIONS

Let G = (N, A) be a directed network with N as the node set and A as the arc set. Let $n = |N|$ and $m = |A|$. The source *s* and the sink *t* are two distinguished nodes of the network. Let $u_{ij} > 0$ denote the capacity of each arc $(i, j) \in A$. Some of the algorithms tested by us require capacities to be integral while others do not. Hence we assume that all u_{ij} are positive integers. This assumption is true for most applications of the maximum flow problem. We further assume that none of the paths from source to sink has infinite capacity. Such a path can be easily detected in O(m) time and would imply an unbounded optimum solution. We further assume that all arcs have finite capacity. There is no loss of generality in this assumption if the network contains no infinite capacity path from source to sink. Under this assumption, the capacity of infinite capacity arcs can be replaced by $\sum_{(i,\, j)\in A: u_{ij} > 0}$. Let $U = \max\{u_{ij} : (i, j) \in A\}$. We define the *arc adjacency* list A(i) of node $i \in N$ as the set of arcs directed out of the node i, i.e., $A(i) : \{(i, k) \in A : k \in N\}$.

A *flow* is a function $x : A \longrightarrow R$ satisfying

$$\sum_{\{j:(j,\, i)\in A\}} x_{ij} - \sum_{\{j:(j\; i)\in A\}} x_{ij} = 0 \text{ for all } i \in N\text{-}\{s, t\}, \tag{1}$$

$$\sum_{\{j:(j\; t)\in A\}} x_{it} = v, \tag{2}$$

and $\quad 0 \le x_{ij} \le u_{ij}$, for all $(i, j) \in A$, (3)

for some $v \ge 0$. The maximum flow problem is to determine a flow x for which v is maximum.

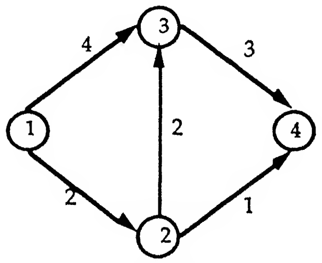
A *preflow x* is a function $x : A \longrightarrow R$ which satisfies (2), (3), and the following relaxation of (1):

$$\sum_{\{j:(j,\, i)\in A\}} x_{ij} - \sum_{\{j:(j\; i)\in A\}} x_{ij} \ge 0 \text{ for all } i \in N - \{s, t\}, \tag{4}$$

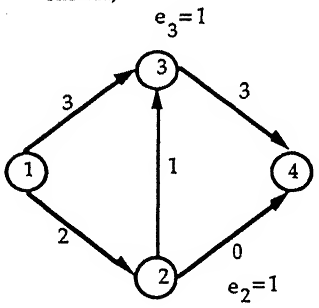
Some of the algorithms described in this thesis maintain a preflow at each intermediate stage. For a given preflow x, we define for each node $i \in N - \{s, t\}$, the *excess*

$$e_j = \sum_{\{j:(j,\, i)\in A\}} x_{ij} - \sum_{\{j:(j\; i)\in A\}} x_{ij}.$$

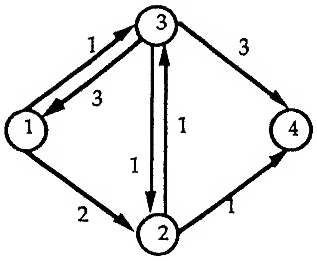
A node with positive excess is referred to as an *active node.* We define the excess of the source and sink nodes to be zero; consequently, these nodes are never active. The *residual capacity* of any arc (i, j)∈ A with respect to a given preflow x is given by $r_{ij} = u_{ij} - x_{ij} + x_{ji}$. The residual capacity of arc (i, j) represents the maximum additional flow that can be sent from node i to node j using the arcs (i, j) and (j, i). The network consisting only of arcs with positive residual capacities is referred to as the *residual network.* Figure 1.1 illustrates these definitions.



a. Network with arc capacities. Node 1 is the source and node 4 is sink. (Arcs with zero capacities are shown.)



b. Network with a preflow x.



c. The residual network with residual arc capacities.

igure 1.1. Illustrations of a preflow and the residual network

1.6 OVERVIEW OF THE THESIS

In Chapter 2, we present a brief survey of the past developments in maximum flow algorithms and summarize the results in previous computational studies. We also describe the data structures used in coding the algorithms, the test problems we have used, and the methodology we have adopted to conduct our study. In Chapter 3, we discuss the augmenting path algorithms and the results of our computational investigations on them. Similarly, we deal with the preflow-push algorithms and the excess-scaling algorithms in Chapters 4 and 5, respectively. In Chapter 6, we present concluding remarks.

CHAPTER 2

LITERATURE SURVEY AND PRELIMINARIES

2.1 INTRODUCTION

The choice of efficient data structures to implement the maximum flow algorithms and the selection of appropriate test problems are two important aspects of computational experiments with algorithms. Apart from these two, in this chapter, we also describe the broad methodology adopted to conduct computational investigations. But, first we briefly summarize the theoretical developments in the maximum flow algorithms and some important results of previous computational studies.

2.2 HISTORICAL BACKGROUND

Theoretical Developments

The maximum flow problem was first studied by Dantzig and Fulkerson [1956], Ford and Fulkerson [1956] and Elias, Feinstein and Shannon [1956] who independently established the max-flow min-cut theorem. Fulkerson and Dantzig [1955] solved the maximum flow problem by specializing the primal simplex algorithm, whereas Ford and Fulkerson [1956] and Elias et al. [1956] solved it by augmenting path algorithms. Since then, researchers have developed a number of algorithms for this problem. The following table summarizes the running times of some of these algorithms.

#	Discoverers	Running Time
1.	Edmonds and Karp [1972]	$O(nm^2)$
2.	Dinic [1970]	$O(n^2m)$
3.	Karzanov [1974]	$O(n^3)$
4.	Cherkasky [1977]	$O(n^2\sqrt{m})$
5.	Malhotra, Kumar and Maheshwari [1978]	$O(n^3)$
6.	Galil [1980]	$O(n^{5/3}m^{2/5})$

7.	Galil and Naamad [1980]; Shiloach [1978]	$O(nm \log^2 n)$
8.	Shiloach and Vishkin [1982]	$O(n^3)$
9.	Sleator and Tarjan [1983]	$O(nm \log n)$
10.	Tarjan [1984]	$O(n^3)$
11.	Gabow [1985]	$O(nm \log U)$
12.	Goldberg [1985]	$O(n^3)$
13.	Goldberg and Tarjan [1986]	$O(nm \log (n^2/m))$
14.	Cheriyan and Maheshwari [1989]	$O(n^2 \sqrt{m})$
15.	Ahuja and Orlin [1989]	$O(nm + n^2 \log U)$
16	Ahuja, Orlin and Tarjan [1989]	(a) $O(nm + \frac{n^2 \log U}{\log \log U})$
		(b) $O(nm + n^2 \sqrt{\log U})$
		(c) $O(nm \log(\frac{n^2\sqrt{\log U}}{m}))$

Table 2.1. Running times of maximum flow algorithms.

Ford and Fulkerson [1956] observed that the labeling algorithm can perform as many as O(nU) augmentations for networks with integer arc capacities. For arbitrary irrational arc capacities, the labeling algorithm can perform an infinite sequence of augmentations and might converge to a value different from the maximum flow value. Edmonds and Karp [1972] suggested two specializations of the labeling algorithm, both with improved computational complexity. They showed that if the algorithm augments flow along a shortest path (i.e., one containing the smallest possible number of arcs) in the residual network, then the algorithm performs O(nm) augmentations. A breadth-first-search of the network will determine a shortest augmenting path; consequently, this version of the labeling algorithm runs in $O(nm^2)$ time. Edmonds and Karp's second idea was to augment flow along a path with maximum residual capacity. They proved that this algorithm performs O(m log U) augmentations. It is shown in Tarjan [1986] how to determine a path with maximum residual capacity in

O(m) time on average; hence, this version of the labeling algorithm runs in $O(m^2 \log U)$ time.

Dinic [1970] independently introduced the concept of shortest path networks, called *layered networks* , and the concept of *blocking flow* in a layered network. Dinic showed how to construct a blocking flow in a layered network by performing at most m augmentations requiring a total of O(nm) time. Dinic's algorithm proceeds by constructing layered networks and establishing blocking flows in these networks. Dinic showed that at each iteration, the length of the layered network increases and after at most n iterations, the source is disconnected from the sink. Consequently, Dinic's algorithm runs in $O(n^2m)$ time.

Researchers have made several subsequent improvements in maximum flow algorithms by developing more efficient algorithms to establish blocking flows in layered networks. Karzanov [1974] introduced the concept of *preflows* in a layered network. (Even [1979] gives a comprehensive description of this algorithm and the paper by Tarjan [1984] simplifies it.) Karzanov showed that an implementation that maintains preflows and pushes flows from nodes with excesses, constructs a blocking flow in $O(n^2)$ time. Malhotra, Kumar and Maheshwari [1978] present a conceptually simpler maximum flow algorithm that runs in $O(n^3)$. This algorithm is subsequently referred to as *MKM algorithm*. Cherkasky [1977] and Galil [1980] presented further improvements of Karzanov's algorithm.

The search for more efficient maximum flow algorithms has led to the development of new data structures for implementing Dinic's algorithm. The first such data structures were suggested by Shiloach [1978] and Galil and Naamad [1980]. Sleator and Tarjan [1983] improved this approach by using a data structure called *dynamic trees*. All of these data structures are quite sophisticated and require substantial overheads. Using a simple scaling technique, Gabow [1985] obtained an algorithm whose time bound under the similarity assumption (i.e., $U = O(n^k)$ for some constant k) was comparable to that of Sleator and Tarjan's algorithm. Recently, Ahuja and Orlin [1991] presented a variation of Gabow's algorithm achieving the same time bound.

A set of new maximum flow algorithms emerged with the discovery of distance labels by Goldberg [1985] in the context of preflow-push algorithms. The distance labels, which implicitly stored layered networks, were easier to manipulate and led to more efficient algorithms. Goldberg [1985] developed the first-in first-out (FIFO) version of the preflow-push algorithm. Goldberg and Tarjan [1986] subsequently developed the generic preflow-push algorithm and the highest label preflow-push algorithm. Goldberg and Tarjan also obtained improved time bounds using dynamic tree data structure in their preflow-push algorithms.

Ahuja and Orlin [1989] improved Goldberg and Tarjan's algorithm using the concept of *excess scaling*. Ahuja, Orlin and Tarjan [1989] developed two improved versions of the excess scaling algorithm. Ahuja, Orlin and Tarjan [1989] also suggested dynamic tree implementations of their algorithms, but the purpose was to improve the worst-case running time, not the running time in practice. Ahuja and Orlin [1991] have investigated the use of distance labels for augmenting path algorithms. They proposed two augmenting path algorithm whose worst-case running time is comparable to that of Dinic's algorithm.

Cheriyan and Hagerup [1989] proposed a randomized algorithm for the maximum flow problem which has an expected running time of O(nm) for all $m \geq n\log^2 n$. Alon [1990] developed a nonrandomized version of this algorithm and obtained a (deterministic) maximum flow algorithm that runs in (i) O(nm) time for all $m = \Omega(n^{5/3}\log n)$, and (ii) O(nm log n) time for all other values of n and m. Cheriyan, Hagerup and Mehlhorn [1990] obtained an $O(n^3/\log n)$ algorithm for the maximum flow problem.

Computational Results

We now summarize the results of the previous computational studies conducted by a number of researchers including Hamacher [1979], Cheung [1980], Glover, Klingman, Mote and Whitman [1979, 1980], Imai [1983], Goldfarb and Grigoriadis [1986] and Derigs and Meier [1989].

Hamacher [1979] tested Karzanov's algorithm versus the labeling algorithm and, as one would expect, found Karzanov's algorithm to be substantially superior to the labeling algorithm. Cheung [1980] conducted an extensive study of maximum flow algorithms including the depth-first-search and breadth-first-search versions of the labeling algorithm, the maximum capacity augmentation algorithm, Dinic's and Karzanov's algorithms. Cheung [1980] ranked these algorithms in the decreasing order of their performance as follows: Dinic's algorithm, Karzanov's algorithm, breadth-first-search labeling algorithm, depth-first-search labeling algorithm, and maximum capacity augmentation algorithm. However, Cheung did not use the most appropriate data structure existing at that time and his results may not hold if algorithms are implemented using better data structures.

Glover, Klingman, Mote and Whitman [1979] conducted another extensive study of maximum flow algorithms using state-of-the-art data structures and considering different network topologies. They tested the primal simplex algorithm, breadth-first-search, depth-first-search and maximum capacity augmentation version of the labeling algorithm, Dinic's algorithms and the MKM algorithm. Their study did not test Karzanov's algorithm. They found that the data structure used and the network topology of problems had a significant effect on the solution times. Roughly speaking, this study found Dinic's algorithm to be the most efficient, followed by MKM and the primal simplex algorithm. Among the labeling algorithms, the breadth-first version was found to be the best.

Imai [1983] performed another extensive study of the maximum flow algorithms and found results that are consistent with those of Glover, Klingman, Mote and Whitman [1979]. His investigations included Karzanov's algorithm and overall he found Karzanov's algorithm to be superior to Dinic's algorithm.

Researchers have also tested implementations of Dinic's algorithm using sophisticated data structures. Imai [1983] tested Galil and Naamad's [1980] data structure, and Sleator and Tarjan [1983] tested their dynamic tree data structure. Both the studies observed that these data structures slowed down the original Dinic's algorithm by a constant factor for all classes of problems considered by them.

Recently, Goldfarb and Grigoriadis [1987] compared the "steepest-edge" version of the primal simplex algorithm with Dinic's algorithm and found the former algorithm to be faster than the latter for some classes of networks. Goldfarb and Grigoriadis, however, used a "space-efficient" version of Dinic's algorithm (personal communication with R. K. Ahuja) and results may be different if a "time efficient" version of Dinic's algorithm is used.

We believe that until recently, Dinic's and Karzanov's algorithms have been the two fastest algorithms for solving the maximum flow problem for most classes of problems. Dinic's algorithm is comparable to Karzanov's algorithm for sparse networks and for dense networks Dinic's algorithm is slower than Karzanov's algorithm.

The first computational investigation of the recent preflow-push algorithms is due to Derigs and Meier [1989]. They tested the FIFO, LIFO and DEQUE (a concatenation of stack and queue) versions of Goldberg's algorithms and suggested strategies to obtain speed-ups in practice. They found that Goldberg's approach with certain modifications suggested by them was faster than some implementations of Dinic's and Karzanov's algorithms by more than one order of magnitude.

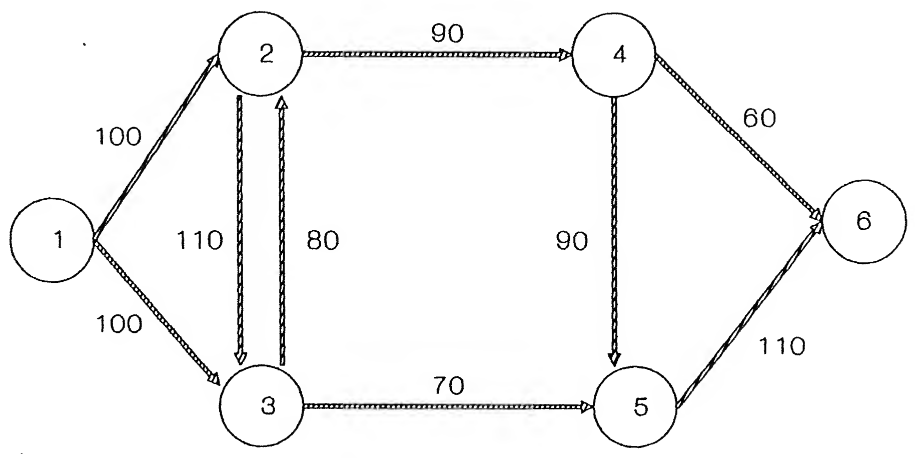
2.3 DATA STRUCTURES

We now describe the data structures we used to implement the maximum flow algorithms. We describe the data structures used to represent networks, and the data structures used to store multiple, singly, and doubly liked lists containing disjoint elements.

The *forward star* and *reverse star representations* are probably the most popular representations to store a network (see, for example, Ahuja, Orlin and Magnanti [1993]). One drawback of this representation is that the information about the forward star of a node (i.e., outgoing arcs from the node) is stored in one array and information about the reverse star of a node (i.e., incoming arcs at a node) is stored in another array. While examining a node, the maximum flow algorithms have to access arcs in both the forward and reverse stars of the node. This leads to additional overheads and significant duplication of statements. Therefore, we did not use this representation. Instead, we used linked list representation of networks that was also used by Imai [1983]. This data structure does not have the above mentioned drawback. Further, since for comparison we used Imai's codes for Dinic's and Karzanov's algorithm, we thought that the use of the same data structure should lead to a fairer comparison of algorithm. It may be noted that Glover et al. [1979] used a similar data structure in their computational testing.

Since the linked list data structure, subsequently referred to as *linked star representation,* is not very popular, we describe this in detail. This data structure uses three arrays of size 2m, namely *STAR, LINK* and *CAP,* and one array to size n, namely *POINT*. Given a network, we form these arrays as follows. We first arrange arcs in any arbitrary order, and then examine arcs in this order. We store the head node of the i-th arc in *STAR(i)* and tail node in in *STAR(m+i)*. We store the capacity of this arc in *CAP(i)* and set *CAP(m+i)* to zero. After forming the *STAR* and *CAP* arrays, we form a linked list of arcs incident at each node i and store it in the *LINK* array. The *POINT(i)* stores the address of the first arc in the linked list. See Figure 2.1 for an illustration of this representation. This is followed by the discussion explaining how all arcs incident at a node can be retrieved in this data structure.

Using this data structure, we retrieve information about arcs incident on a node as follows. Suppose we wish to determine all arcs incident on node 4. We look at *POINT(4)* which is equal to 8. We look at the 8-th place in the arrays and find that



	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
STAR	2	4	3	3	2	5	6	5	6	1	2	1	2	3	3	5	4	4
CAP	100	90	100	110	80	70	110	90	60	0	0	0	0	0	0	0	0	0
LINK	3	4	0	10	6	12	15	9	11	14	0	13	0	0	17	0	0	16

	1	2	3	4	5	6
POINT	1	2	5	8	7	18

Figure 2.1. Linked Star Representation

STAR(8) = 5, *CAP(8)* = 90 and *LINK(8)* = 9. Hence the first arc is (4, 5) with residual capacity equal to 90. The *LINK(8)* is a pointer to the next arc in the adjacency list of node 4. Since *LINK(8)* = 9, we look at the 9-th place in the arrays. We find that *STAR(9)* = 6, *CAP(9)* = 60 and *LINK(9)* = 11. Hence the second arc is (4, 6) with the residual capacity equal to 60. The LINK(9) = 11 is a pointer to the next arc in the adjacency list of node 4. It may be noted that whenever the pointer to an arc is greater than m (is our case, m = 9), then the corresponding arc is an incoming arc; otherwise it is an outgoing arc. We look at the 11-th place in arrays and find that *STAR(11)* = 2, *CAP(11)* = 0 and *LINK(11)* = 0. Hence the third arc in the adjacency list of node 4 is (2, 4) with zero residual capacity. Since *LINK(11)* is zero, we have reached the end of the adjacency list.

Representing Singly Linked Lists of Disjoint Elements

The preflow-push algorithms store the sets of disjoint elements *LIST(1)*, *LIST(2)*, . . . , *LIST(n)*. The value of each element lies between 1 and n, and elements are added or deleted from the front of the lists. We store these sets as *linked stacks*. Suppose that n = 6; and *LIST(1)* ={5, 4), *LIST(3)* = {1, 2, 3}, *LIST(4)* = {6} and *LIST(2)* = *LIST(5)* = *LIST(6)* = ϕ. We store all of these sets using two n-size arrays, namely *FIRST* and *LINK*, as follows.

Index	FIRST	LINK
1	5	2
2	0	3
3	1	0
4	6	0
5	0	4
6	0	0

Figure 2.2 Storing singly linked lists of disjoint elements.

In this scheme, *FIRST(k)* stores the first element in the set *LIST(k)*. If *FIRST(k)* = 0, then *LINK(k)* is empty; otherwise it is non-empty. For example, *FIRST(1)* = 5. Hence the first element in *LIST(1)* is 5. We then look at *LINK(5)* which is 4. Hence the second element in *LIST(1)* is 4. We then look at *LINK(4)* which is 0, indicating the end of the list. Hence the second element in *LIST(1)* = {5, 4}. Suppose we wish to add an element p to *LIST(k)*. Observe that due to the non-disjoint nature of the lists, p must not be present in any of the lists. We perform the statements *LINK(p)* = *FIRST(k)* and *FIRST(k)* = p. To delete an element p from *LIST(k)*, we perform *FIRST(k)* = *FIRST(LINK(k))*.

The singly linked list storage scheme allows us to delete the first element of a set in O(1) time. If we want to delete any arbitrary element from a set in O(1) time, then we need to represent the set as a doubly linked set. We describe next a data structure to store n sets of disjoint elements.

Representing Doubly Linked Lists of Disjoint Elements

Suppose again that we wish to store the sets *LIST(1), LIST(2), . . . , LIST(n)*, whose elements are disjoint and vary between 1 and n. We consider the same example with n = 6, *LIST(1)* ={5, 4}, *LIST(3)* = {1, 2, 3}, *LIST(4)* = {6} and *LIST(2)* = *LIST(5)* = *LIST(6)* = θ. We store these sets using three n-size arrays, namely *FIRST, RLINK* and *LLINK*, as follows.

Index	FIRST	RLINK	LLINK
1	5	2	0
2	0	3	1
3	1	0	2
4	6	0	5
5	0	4	0
6	0	0	0

Figure 2.3. Storing doubly linked lists of disjoint elements.

2.4 NETWORK GENERATORS

An algorithm could perform well when tested on some networks and poorly on others. Considering our primary objective, we had to chose networks such that an algorithm's performance on it could give sufficient insight into its general behavior. In maximum flow literature, no particular type of network has been favored for empirical analysis. We tried several networks before choosing Random Layered networks and Random Grid networks for our testing. We found that pure random networks were very easily solved by all the algorithms. We wanted to investigate the algorithms on sufficiently hard networks. Hence, we investigated structured networks and settled for the two networks mentioned above of which the density of one can be varied and the other is sparse. We also tried NETGEN developed by Klingman et al. [1974] but found it to be far more easier than our networks.

Random Layered Networks

This network generator produces networks with topological structure of a layered network; hence we name it the random layered network generator. In this and the following network generation scheme, namely random grid network, we need two parameters: width W and length L. For given values of W and L the network has WL +2 nodes of which WL nodes are arranged in L layers; these layers are numbered from 1 to L. Each of these layers contains W nodes. The source node is in layer 0 and the sink node is in layer (L+1). See Figure 2.4 for an example of a random layered network. This arrangement of nodes assigns two co-ordinates, level and layer, with each node which specify its position with respect to other nodes. For example, node 3 is in layer 1 and at level 3, and node 4 is in layer 2 and at level 1.

Each arc in this network connects a node to another node in the next layer. The source node is connected to each node in layer 1, and each node in layer L is connected to the sink node. Rest of the arcs are generated randomly using another parameter $p \leq W/2$, which represents the average out-degree of a node. For each node i (say, which is layer k and level l) we select a number, say w, from the uniform distribution in the range [1, 2p–1] and then generate w arcs emanating from node i whose head nodes are randomly selected from nodes in the layer (l+1). Capacities of the arcs are assigned in the following manner. The capacities of the source and sink arcs, i.e., arcs incident to the source and sink nodes, are set to a large number. This essentially amounts to creating W sources and W sinks and makes the maximum flow problems harder. Capacities of other arcs are randomly selected from a uniform distribution in the range [500,10000].

In our experiments we considered networks with different sizes. Two parameters determined the size of the networks: number of nodes (n) and the average out-degree (p). For the same number of nodes we tested different combinations of width (W) and length (L) and chose L/W=2; various values of L/W gave similar results unless the network was sufficiently long or wide to influence the results for even the largest networks we tested. We felt that L/W=2 was a fair representative length to width ratio. For given values of n and p, we generated twenty different problems by varying the seed to the random number generator, and the average values of the statistics were analyzed. For each n we generated problems with p=4, 6, 8, and 10. The following table specifies the values of n we selected and for each set of n and p, the values of W and L we used. Observe that we give the approximate round figures of the number of nodes and arcs, since it is not possible to generate different networks with exactly same number of nodes and arcs. We believe that it does not introduce any significant error in the results.

	n	WIDTH (W)	LENGTH (L)
1	500	16	31
2	1000	22	45
3	2000	32	63
4	3000	39	77
5	4000	45	89
6	5000	50	100
7	6000	55	109
8	7000	59	119
9	8000	64	125
10	9000	67	134
11	10000	71	141

Table 2.2. Network dimensions.

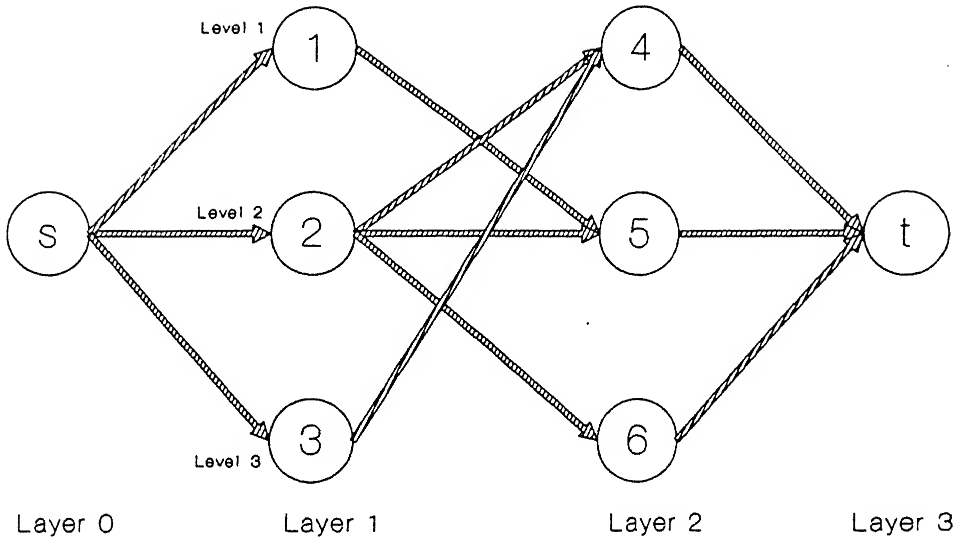


Figure 2.4. Random Layered Network.
W=3, L=2, p=2

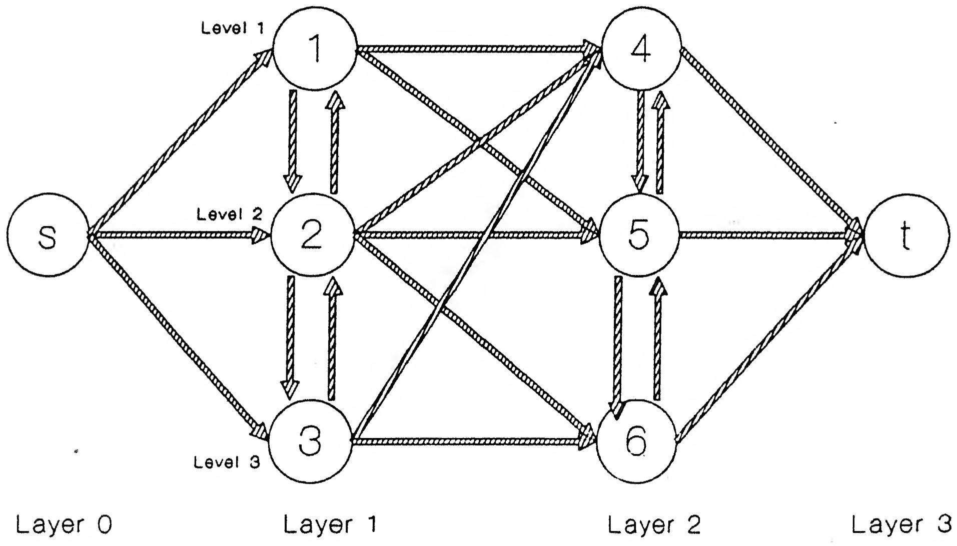


Figure 2.5. Random Grid Network.
W=3, L=2

Random Grid Networks

The random grid networks also use the parameters W and L, and have WL+2 nodes which are arranged in layers in the same manner as in the random layered networks. The topological structure of this network is completely determined by W and L. The source code is connected to all nodes in layer 1 and all nodes in layer L are connected to the sink node. Suppose a node i is in layer k at level *l*. Then node i is connected to the nodes at level (*l*–1) and (*l*+1) in layer k, and the nodes at level (*l*–1), l, (*l*+1) in layer (k+1). If node i is a boundary node (i.e., *l*=1 or W), then some of these arcs may not be present. Figure 2.5 shows a random grid network.

We generate the arc capacities in the following manner. The capacities of the source and sink arcs are set to a large number. For the non-source and non-sink arcs, the capacity is selected randomly from a uniform distribution in some range. For arcs with their end points in the same layer, the range is [200,10000]; and for arcs with their end points in different layers, the range is [500,10000]. We select different ranges because we think that this scheme generates more difficult networks.

We generated random grid networks of different sizes by using L/W=2 and varying n from 500 to 10000, as for the random layered network and we used the same values of W and L for the grid network as in Table 2.2. We solved twenty problems for each n by changing the input seed to the network generator and took the average of these runs.

2.5 METHODOLOGY FOR COMPUTATIONAL INVESTIGATION

Ahuja and Orlin [1993] describes a method to perform computational studies on algorithms. The methodology adopted in this study of the empirical behavior of the maximum flow algorithms is as described in this paper. For the sake of completeness, we briefly summarize the essential ideas of their paper.

A large number of computational studies have been reported in the operations research literature. A typical study tests more than one algorithm for a specific network flow problem and generally consists of the following steps:

(i) write a computer program, in the same language, for each algorithm to be tested;

(ii) use pseudo-random network generators to generate random problem instances with selected combinations of input size parameters (e.g., nodes and arcs);

(iii) run computer programs and note the CPU times for the different algorithms on the data obtained by the network generators;

(iv) declare the algorithm that takes the least amount of CPU time as the 'winner' (if different algorithms are faster for different input size parameters, then report this fact too).

Over-dependance on CPU times, as the primary measure of performance has its disadvantages. The CPU times depend greatly upon subtle details of the computational testing environment and the test problems such as: (i) the chosen programming language, compiler, and computer; (ii) the implementation style and the skill of the programmer; (iii) network generators used to generate the random test problems; (iv) combinations of input size parameters; and (v) the particular system environment, e.g., a time-sharing system. Hence CPU times are difficult to replicate. Moreover, CPU time is an aggregate measure of performance, and does not provide much insight into an algorithm's behavior.

Ahuja and Orlin [1993] suggest an approach based on representative operation counts. In this approach, an algorithm (or, the program code) is decomposed into a number of operations requiring $\theta(1)$ time, and of these a small set S of operations is selected satisfying the property that each execution of an operation not in S can be charged to an execution of some operation in S. The set S of operations is called a *representative set of operations*. Then, during program executions if, along with the CPU times, we note the counts of representative operations, they can be used to obtain valuable information about an algorithm's behavior. These representative operation counts allow us to perform the following tasks:

(i) Representative operation counts allow us to identify the asymptotic bottleneck operations of the algorithm, i.e., operations that progressively consume a larger share of the computational time as the problem size increases.

(ii) Representative operation counts permit us to determine lower and upper bounds on the asymptotic growth rate in computational time as a function of the problem size.

(iii) Representative operation counts provide more guidance and insight about comparing two algorithms.

(iv) We can use the statistical methodologies to estimate the CPU time on a computer as a linear function of the representative operation counts. This estimate of the CPU time is referred as *Virtual CPU time*. The virtual CPU time can permit researchers to run their codes on different computers using different languages and compare the running times as if all the experiments had been carried out on a common computer.

Representative Operation Counts

Let *A* be computer program for solving some problem and I be an instance of a problem. The computer program consists of a finite number, K, of lines of code say a_1, a_2, ..., a_K. We assume that the program is written such that each line of code gives O(1) instructions to the computer, and that each instruction requires O(1) units of time for execution. So, each line of code requires $\theta(1)$ time units since its execution time is bounded from both above and below by a constant number of units. For a given instance I of the problem, let $\alpha_k(I)$, for k=1 to K, be the number of times that the computer executes line k of this computer program. Let CPU(I) denote the CPU time of the computer program on instance I. Hence it is implied that

$$CPU(I) = \theta(\Sigma_{k=1}^{k} \alpha_k(I))$$

But it is not really necessary to count the number of executions of each line of code. We can keep track of a small number of lines of code, which we call the "representative operation counts". Let S denote a subset of {1,...,K}, and let as denote the set $\{a_i: i \in S\}$. W say that a_s is a representative set of lines of code of a program if for some constant c,

$$\alpha_i(I) \leq c\ (\Sigma_{k \in S}\ \alpha_k(I)).$$

for every instance I of the problem and for every line a_i of code, i.e., the time spent in executing line a_i is dominated (up to a constant) by the time spent in executing the lines of code in a_S. Hence, if S is a representative set of lines of code, then

$$CPU(I) = \theta(\Sigma_{k \in S}\ \alpha_k(I)).$$

This methodology identifies a representative set of lines of code, and during the empirical investigations keeps track of the representative operation counts for each instance involved. The selected representatives need not refer specifically to one line of code, they might be described over several lines of code. Finding the representative set of operations is not difficult and there are a wide number of possible correct choices. We can keep a record of some other counts that will be helpful in assessing an algorithm's behavior.

We can use the representative operation counts to obtain several conclusions.

(a) Identifying Asymptotic Bottleneck Operations

An operation is a 'bottleneck operation' for an algorithm if it consumes a significant percentage of the execution time on at least some fraction of the problems tested. An operation is an asymptotic nonbottleneck operation if its share in the computational time becomes smaller and approaches zero as the problem size increases. Otherwise, an operation is an asymptotic bottleneck operation. The asymptotic bottleneck operations determine the running times of the algorithm for sufficiently large problem sizes. Since, the representative operation counts provide both upper and lower bounds on the number of operations an algorithm performs, this set must also contain at least one asymptotic bottleneck operation. In the absence of any formal and rigorous method to find an asymptotic bottleneck operation, we use the following method.

Let A and B be the representative operations for the algorithm being studied. Let, $\alpha_S(I) = \alpha_A(I) + \alpha_B(I)$. We plot $\alpha_A(I)/\alpha_S(I)$ and $\alpha_B(I)/\alpha_S(I)$ for increasingly larger problem instances I and look for a trend. The operation having an increasing share in the sum of the representative operation counts is the asymptotic bottleneck operation in the algorithm.

(b) Comparing two algorithms

Let $\alpha_{S1}(k)$ and $\alpha_{S2}(k)$ be the total number of expected operations performed by the algorithms AL_1 and AL_2 on instances of size k. We say that AL_1 is superior to the algorithm AL_2 if

$$\lim_{k \to \infty} \alpha_{S1}(k)/\alpha_{S2}(k) = 0.$$

(c) Virtual running times

The virtual running time of an algorithm is a linear estimate of its CPU time. The virtual running time V(I) of an algorithm to solve problem instance I is given by

$$V(I) = c_A\, \alpha_A(I) + c_B\, \alpha_B(I),$$

where c_A and c_B are constants selected so that V(I) is the best possible estimate of the CPU time given by the algorithm's actual running time CPU(I) on the instance I. One of the possible ways to estimate c_A and c_B is to use (multiple) regression analysis. We consider the points (CPU(I), c_A, c_B) generated by solving various individual instances and use regression analysis to determine the values of constants c_A and c_B that minimizes the expression $\Sigma_I(\text{CPU}(I) - V(I))^2$. To obtain an idea of the goodness of this fit, we plot the ratio V(I)/CPU(I) for all the data points.

We can also estimate the proportion of time that an algorithm spends on different representative operations. We can identify not only the asymptotic bottleneck operation, but also the bottleneck operations for different sizes of problem instances. We report this study for several algorithms in the next chapters.

(d) Estimating growth rate of bottleneck operations

To estimate the growth rate of a bottleneck operation, say α_A, we determine an appropriate functional form for estimating the counts for the bottleneck operation. For example, we choose the functional form to be n^γ for some choice of a growth parameter γ. A more common approach would be to choose a function of both n and m; however, we consider each network density separately, and for each network density $d = m/n$, the ratio of m to n is fixed. So a function of m and n reduces to a function of n. We now plot $\gamma = \log(\alpha_A)/\log(n)$, for all values of n and d. Considering large values of n, we can get a fair estimate of the lower and upper bounds on α_A.

(e) Additional insights into an algorithm

Representative operation counts can be used to obtain several additional insights into an algorithm. There are several instances in this study where this is done.

2.6 ENVIRONMENT USED FOR TESTING

All the programs were coded in FORTRAN 77. They were run on the Convex mini super computer under the Convex OS 10.0.5 using the Convex Fortran compiler V 7.0.1 in a time-sharing environment. Efforts were made to run the programs under similar conditions of load on the computer resources. The CPU time taken by the programs were noted using a standard available time function having a resolution of 1 microsecond. The time reported does not include the time taken for input or output. It does include the time taken to initialize the variables.

CHAPTER 3

AUGMENTING PATH ALGORITHMS

3.1 INTRODUCTION

Augmenting path algorithms incrementally augment flow along paths from the source node to the sink node while maintaining the mass balance at every node of the network other than the source and sink nodes. The labeling algorithm is a special implementation of the generic augmenting path algorithm and it runs in pseudopolynomial time. Several studies in the past have shown that it is slow for large size problems. Augmenting flow along shortest augmenting paths and augmenting in large increments of flow are two strategies that have reduced the number of augmentations in the labeling algorithm and have provided speed-ups in worst-case and empirical performance. Of the several augmenting path algorithms suggested in the past, we studied the shortest augmenting path algorithm, Dinic's algorithm, and the capacity scaling algorithm. We describe these algorithms and their implementations briefly and then report the results of our computational investigations.

3.2 THE SHORTEST AUGMENTING PATH ALGORITHM

A directed path from the source to the sink in the residual network is called an *augmenting path*. We define the residual capacity of an augmenting path as the minimum residual capacity of any arc in the path. Observe that by the definition of an augmenting path, its capacity is always positive. Hence presence of an augmenting path implies that additional flow can be sent from source to sink. The generic augmenting path algorithm due to Ford and Fulkerson [1956] is based on this simple idea. The algorithm proceeds by identifying augmenting paths and augmenting flows on these paths until no such path exists.

A natural specialization of the augmenting path algorithm is to augment flow along a "shortest path" from the source to the sink, defined as a directed path in the residual network consisting of least number of arcs. If we augment flow along a shortest path, then the length of a shortest path either stays the same or increases. Moreover, within m augmentations, the length of a shortest path is guaranteed to increase. Since no path contains more than n-1 arcs, this rule guarantees that the number of augmentations is at most (n-1)m. This algorithm is called the *shortest augmenting path algorithm.*

A shortest augmenting path can be easily determined by performing a breadth-first-search of the residual network. This approach would take O(m) steps per augmentation, in the worst case as well as in the practice. We now discuss the concept of distance labels, which allows us to reduce the average worst-case time per augmentations to O(n).

Distance Labels

A *distance function* $d : N \rightarrow z^t$ with respect to the residual capacities r_{ij} is a function from the set of nodes to the non-negative integers. We say that a distance function is *valid* if it satisfies the following two conditions:

C1. $d(t) = 0$;

C2. $d(i) \leq d(j) + 1$ for every $(i, j) \in A$ with $r_{ij} > 0$.

We refer to d(i) as the distance label of node i and condition C2 as the validity condition. The following observations can be easily proved about distance labels.

Observation 3.1. If distance labels are valid then each d(i) is a lower bound on the length of the shortest (directed) path from i to t in the residual network.

Observation 3.2. If $d(i) \geq n$, then there is no path from source to sink in the residual network.

Admissible Arcs and Admissible Paths

We call an arc (i, j) in the residual network as *admissible* if it satisfies $d(i) = d(j) + 1$. An arc that is not admissible, is called *inadmissible.* We call a path from s to t consisting entirely of admissible arcs as an *admissible path.* The shortest augmenting path algorithm uses the following observation.

Observation 3.3. An admissible path is a shortest augmenting path from source to sink.

Since every arc (i, j) in an admissible path P is admissible, it satisfies (i) $r_{ij} > 0$, and (ii) $d(i) = d(j) + 1$. Property (i) implies that P is an augmenting path. Property (ii) implies that if P consists of k arcs, then $d(s) = k$. As d(s) is a lower bound on any length of any path from source to sink (from Observation 3.1), the path P must be a shortest path.

The shortest augmenting path algorithm proceeds by augmenting flows along admissible paths. It obtains an admissible path by successively building it up from scratch. The algorithm maintains a *partial admissible path,* i.e., a path from s to some node i consisting solely of admissible arcs, and iteratively performs *advance* or *retreat* steps at the last node (i.e., tip) of the partial admissible path, called the *current-node.* If the current-node i has an admissible arc (i, j) then we perform an *advance* step and add arc (i, j) to the partial admissible path; otherwise we perform a *retreat* step and backtrack by one arc. We repeat these steps until the partial admissible path reaches the sink node at which time we perform an augmentation. We repeat this process until the flow is maximum. A formal description of the algorithm is given below.

initialize. Perform the breadth-first-search of the residual network starting with the sink node to compute the exact distance labels d(i). Let P = ϕ and i = s. Go to *advance(i).*

advance(i) If there does not exist an admissible arc (i, j) then go to *retreat(i).* If there exists an admissible arc (i, j) then set *pred(j):* = i and P: = P$\cup${i, j}. If j = t then go to *augment;* else replace i by j and repeat *advance(i).*

retreat(i). Update d(i): = min{d(j)+1 : $r_{ij} > 0$ and (i, j)$\in$ A(i)}. (This operation is called a *relabel* operation.) If d(s) $\geq$ n, then STOP. If i = s then go to *advance(i);* else delete (*pred(i),* i) from P, replace i by *pred(i)* and go to *advance(i).*

augment. Let Δ: = min{r_{ij}: (i, j)$\in$P. Augment Δ units of flow along P. Set P: = ϕ, i: = s and go to *advance(i).*

The shortest augmenting path algorithm uses the following data structure to identify admissible arcs emanating from a node in the *advance* steps. Recall that we maintain the arc list A(i) comprising of all arcs emanating from node i. We can arrange arcs in this list arbitrarily, but the order once decided remains unchanged throughout the algorithm. Each node i has a *current-arc,* which is an arc is A(i) and is the next candidate for testing admissibility. Initially, the *current-arc* of node i is the first arc in A(i). Whenever the algorithm has to find an admissible arc emanating from node i, it tests whether its *current-arc* is admissible. If not, then it makes the next arc in the arc list as the *current-arc.* The algorithm repeats this process until either it finds an admissible arc or reaches the end of the arc list. In the later case, the algorithm declares that there is no admissible arc in A(i). It then performs a relabel operation and again sets the *current-arc* of node i to the first arc in A(i).

The algorithm uses the following properties to obtain a bound on its worst-case complexity.

Lemma 3.1. (a) The algorithm maintains valid distance labels at each step.

(b) Each relabel step increases the distance label of a node by at least one unit.

(c) During the intermediate steps of the algorithm, $d(i) \le n$ for each i.

(d) The algorithm performs at most $n\,m/2$ augmentations.

Proof. Omitted. ■

Using the above results, it can be shown that the shortest augmenting path algorithm runs in $O(n^2m)$ time in the worst-case. However, from an empirical viewpoint these are two bottleneck operations in the algorithm.

Relabeling Time: Lemma 3.1 implies that any node is relabeled at most n times. As relabeling of a node i takes $O|A(i)|$ time, the total relabeling time is $O(\sum_{i\in A} |A(i)|) = O(nm))$. The time to identify admissible arcs during advance steps is also O(nm) because between two consecutive scanning of arcs in the set A(i) for some node i, d(i) strictly increases. In the subsequent discussion, we include the time of identifying admissible arcs in the relabeling time. Each *retreat* step relabels a node, hence the number of retreat steps is $O(n^2)$.

Augmentation Time: It follows from Lemma 3.1(d) that the augmentation time is $O(n^2m)$ because each augmentation takes O(n) time. The number of *advance* steps performed by the algorithm is also $O(n^2m)$. We shall subsequently include the number of *advance* steps in the augmentation time.

We now describe some techniques that speed up the augmenting path algorithm in practice.

The algorithm as described earlier terminates only when $d(s) \ge n$, even though the maximum flow would have been established much earlier. For example, consider the maximum flow problem described in Figure 3.1. Assume that arcs in A(i) are arranged as {(1, 2), (1, 3), (1, 4), (1, 5), (1, 6)}. It can be verified that the shortest augmenting path algorithm would establish a maximum flow within five augmentations and at the end of the last augmentation the distance labels would be d = (3,4,4,4,4,2,1,1,1,1,1,0). Then the algorithm successively increases the distance labels of nodes 6, 1, 2, 3, 4, 5 in this order, each time by two units, until $d(i) \geq 12$. This phenomenon is quite pervasive among the maximum flow problems. We describe a simple technique, discovered independently by Ahuja and Orlin [1989] and Derigs and Meier [1989] that detects such a situation quickly and terminates the algorithm.

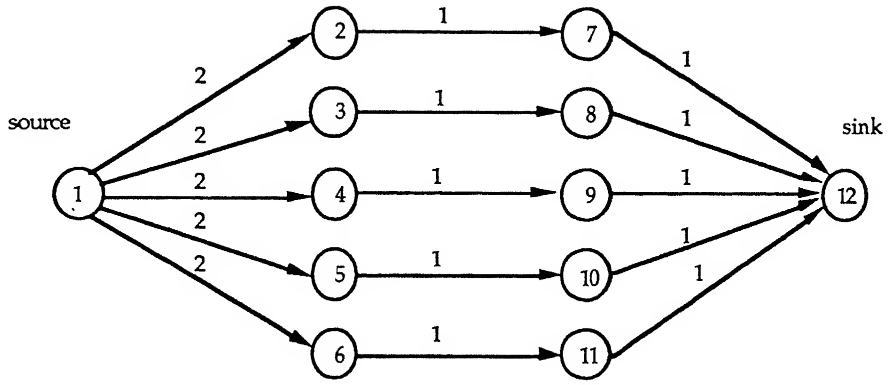


Figure 3.1. A bad example for the shortest augmenting path algorithm.

In this technique, we maintain an array, *numb,* which is defined for the indices 0 to n-1. The value *numb(k)* stores the number of nodes with distance label equal to k. The algorithm initializes this array while computing the initial distance labels using the breadth-first-search. At this point, the positive entries in the array *numb* will be arranged consecutively. Subsequently, whenever the algorithm increases the distance label of a node from k_1 to k_2, it subtracts 1 from *numb* (k_1) adds 1 to *numb* (k_2), and checks whether *numb* $(k_1) = 0$. If *numb* (k_1) is found to be zero then the algorithm terminates. It can be shown that the cut $[S, \bar{S}]$ defined as $S = \{i \in N: d(i) > k_1)$ and $\bar{S} = \{i \in N: d(i) < k_1)$ is a minimum cut which established the correctness of this technique. The reader can verify that if this technique is applied to the maximum flow problem in Figure 3.1, then the shortest augmenting path algorithm would terminate immediately after the last augmentation.

The algorithm as described first identifies an augmenting path, then determines its capacity and then augments flow on this path. However, the algorithm, as implemented, maintains for each node i in the partial admissible path, a *capacity label* representing the residual capacity of the path from source to node. When the partial admissible path is an admissible path then the capacity label of sink is the residual capacity of the admissible path.

Suppose that the algorithm performs an augmentation over a path P and saturates some arcs in the path. Let (p, q) be the saturated arc which is closest to the source. Let P_1 be the segment of path P from source to node p. The algorithm as described would perform the next *advance* step at the source, and since every arc (i, j) in the path segment P_1 is admissible and arc (i, j) is the current arc of node j, the algorithm would reconstruct the partial admissible path P_1. The improved algorithm, during each augmentation locates the arc (p, q) and performs the next *advance* step at node p. This change alone reduces the number of *advance* steps by almost a factor of half.

The last technique, we describe now, reduces the arc scanning time during relabel operations. Recall that between two consecutive updates of a distance label d(i), the algorithm scans arcs in A(i) twice — once while locating admissible arcs emanating from node i and second while relabeling. The idea is to compute an approximate value of (in fact, a lower bound on) the next distance label of node i while locating admissible arcs and hence avoiding the second scan. We do it as follows. After each distance update of node i we set an index *second(i)* = d(i)+2. Further, while locating admissible arcs, if an arc (i, j) is found to be inadmissible because d(i) < d(j)+1, we execute the statement *second(i)* = min {second(i), d(j)+1). When the entire arc list has been scanned we set d(i) = *second(i)*.

3.3 DINIC'S ALGORITHM

Dinic's algorithm is another popular algorithm that uses shortest flow augmenting paths from the source to the sink. This algorithm iteratively constructs layered (or referent) networks and establishes blocking (or maximal) flows in them.

For $i \in N$, let l be a function on G such that $l(i)$ denotes the length of the shortest augmenting path from node i to sink t in G(x). For $i \in N-\{t\}$, if there is no flow augmenting path from i to t, $l(i)=\infty$. Also, $l(t)=0$. A layered network, denoted by G'=(N', A') is defined with respect to a given flow x. It consists of all those nodes in G for which $l(i) \le l(s)$ and all those arcs in G for which (i) if (i,j) is a emanating arc from node i, $l(i) - l(j) = 1$ and $x_{ij} \le c_{ij}$, (ii) if (j,i) is an entering arc on node i, $l(i) - l(j) = 1$ and $x_{ij} \ge 0$. Some points clearly evident about layered networks are:

(i) Every directed path from the source node to the sink node in the layered network is a shortest path in G'.

(ii) The length of every augmenting path is $l(s)$, which we refer as the length of the layered network. By definition, for all $i \in N'$, $l(i) <= l(s)$.

(iii) Some arcs in A' are not contained in any path from the source to the sink. We can delete these arcs from the layered network.

(iv) A layered network is acyclic and layered, i.e., the set N' of nodes in the layered network are partitioned into layers of nodes $N'_0, N'_1, \ldots, N'_{l(s)}$. Layer k contains the nodes which are at a distance k from the sink. For every arc (i, j) in A', $i \in N'_k$ and $j \in N'_{k-1}$, for some k.

Dinic's algorithm performs a depth first search to identify an augmenting path from the source to the sink in the layered network. From (ii) and (iv) above, we can determine an augmenting path in a layered network in O(n) time. Each augmentation saturates at least one arc in the layered network and there is no augmenting path in the layered network after at most m augmentations. This stage is called a blocking or maximal flow. Hence, a blocking flow can be established in a layered network in O(nm) time.

Starting with an appropriate feasible flow x in G = (N, A), which may be a zero flow, Dinic's algorithm tries to send the maximum flow to the sink by repeating the following steps:

(i) Construct the layered network G'=(N', A') from network G=(N, A) with flow x.

(ii) Find a blocking flow x' in G' and update the layered network.

This algorithm terminates when while constructing a new layered network, it finds that the source is not connected to the sink. Each execution of (i) and (ii) is called a phase. Since the length of the layered network strictly increases whenever a new phase begins, and the length of the layered network can at most be n-1, a maximum of n-1 layered networks are constructed and a maximum flow can be established in $O(n^2m)$ time.

We have used the code developed by Imai [1983] to study the Dinic's algorithm. It uses the *subreferent* method of Glover et al. [1979] to represent the layered network. The function *l* is computed by a simple breadth-first search. Arcs leaving i, for $i \in N'$, are found using *l* because as long as we stretch flow-augmenting path starting from s by using arcs in G', nodes that are reachable from s are necessarily contained in N'. At the beginning, all nodes are unlabeled. Starting with s, we scan the arcs incident to a labeled node to search for an unlabeled node j. Node j is assigned a label k where k is the number of the arc using which it was labeled ($1 \leq k \leq 2|A|$) and we resume the search from j. All the labels are not erased after augmenting a flow. The node i, on flow augmenting path P, is found upto which flow can still be pushed after augmenting flow to the sink. In the next depth-first search, the labels given to the nodes that lie on the part of P from s to i are reused and we start searching from i.

The representative operations for Dinic's algorithm are

Augmentation time: The augmentation time is $O(n^2m)$.

Arc scans for constructing layered networks: Since there can be atmost n-1 layered networks, this time is O(nm).

An array of size |N| is used to represent the function *l*. To execute the depth-first search, two arrays of size |N| are used. One is used to record the labels and another to record the arc for each node that should be scanned next on G'.

3.4 CAPACITY SCALING ALGORITHM

In this section we study the capacity scaling algorithm for the maximum flow problem. This algorithm was originally suggested by Gabow [1985]. Ahuja and Orlin [1991] subsequently developed a variant of this approach which is better suited from empirical standpoint. We therefore tested this variant in our computational study.

The essential idea behind the capacity scaling algorithm is to augment flow along a path with *sufficiently large* residual capacity so that the number of augmentations is *sufficiently small*. The capacity scaling algorithm uses a parameter Δ and with respect to a given flow x defines the Δ*-residual network* as a network whose every arc has a residual capacity at least Δ. We denote the Δ*-residual network* by $G(x, \Delta)$. The capacity scaling algorithm works as follows.

```
algorithm capacity scaling;
begin
    x := 0;
    $\Delta := 2^{\lfloor \log U \rfloor}$;
    while $\Delta \geq 1$ do
    begin
        while $G(x, \Delta)$ contains a path from node s to node t do
        begin
            identify a path P in $G(x, \Delta)$;
            $\delta := \min \{r_{ij} : (i, j) \in P\}$;
            augment $\delta$ units of flow along P and update $G(x, \Delta)$;
        end;
        $\Delta := \Delta / 2$;
    end;
end;
```

Figure 3.2 The capacity scaling algorithm.

We call a phase of the algorithm during which Δ remains constant as the *Δ-scaling phase*. In the *Δ-scaling phase,* each augmentation carries at least Δ units of flow. The algorithm starts with $\Delta = 2^{\lfloor \log U \rfloor}$ and halves its value in every scaling phase until $\Delta = 1$. Hence the algorithm performs $1 + \lfloor \log U \rfloor = O(\log U)$ scaling phases. Further, in the last scaling phase, $\Delta = 1$ and hence $G(x, \Delta) = G(x)$. This establishes that the algorithm terminates with a maximum flow.

The efficiency of the algorithm depends upon the fact that it performs at most 2m augmentations per scaling phase (see Ahuja and Orlin [1991]). If we use any search technique to locate augmenting paths in $G(x, \Delta)$ then the capacity scaling algorithm would run is $O(m^2 \log U)$ time. We can, however, improve this by employing the shortest augmenting path algorithm within each scaling phase. Notice that the bottleneck time in the shortest path algorithm is the augmentation time $O(n^2m)$ which is due to the fact that the algorithm performs $O(nm)$ augmentations. If this algorithm is employed to solve the scaled problem, it would perform $O(m)$ augmentations and would run in $O(nm)$ time. As these are $O(\log U)$ scaling phases, the overall running time of the capacity scaling algorithm is $O(n\, m \log U)$.

There are two following bottleneck operations in the capacity scaling algorithm from empirical viewpoint.

Relabeling Time. The algorithm takes $O(nm)$ time for relabeling of nodes and identifying admissible arcs in each scaling phase. Overall this time is $O(nm \log U)$.

Augmentation Time. As observed earlier, the augmentation time is also O(nm log U) over the entire algorithm.

Notice that when compared to the shortest augmenting path algorithm, the capacity scaling algorithm reduces the augmentation time from O(n^2m) to O(nm log U) but increases the relabeling time from O(nm) to O(nm log U). Though this is a major improvement from worst-case point of view, we shall see that it substantially worsens algorithms running time in practice.

To implement the capacity scaling algorithm, we use the implementation of the shortest path algorithm described in Section 3.2 as a subroutine. Further, the capacity scaling algorithm does not maintain the Δ-residual network explicitly. The algorithm maintains the whole residual network and if while scanning an arc it finds that its residual capacity is less than Δ, then it ignores the arc.

3.5 COMPUTATIONAL RESULTS FOR SHORTEST AUGMENTING PATH ALGORITHM

As observed in Section 3.2, the arc scans for augmentations and the arc scans for node relabels are the representative operations for the shortest augmenting path algorithm. We counted these and the number of augmentations and relabels themselves.

The arc scans for augmentations increase fairly uniformly with the increase in the size of the random layered network as observed in Figure 3.3. The arc scans for node relabels vary erratically while showing, on the whole, an increasing trend as seen in Figure 3.4. Both increase, in general, with the increase in the density of the random layered network. We can see in Figure 3.7 that their count for the random grid network has a smooth increasing slope with increasing size of the network.

To determine the asymptotic bottleneck operation, we sum the arc scans for augmentations and arc scans for relabels and plot the share of each in this sum. For both, random layered network and random grid network, the share of relabel time increases with the size of the network [see Figures 3.5, 3.6 and 3.8]. Hence, we observe that arc scans for node relabels is the asymptotic bottleneck operation for the shortest augmenting path algorithm. For the random layered network, the augmentation time dominates the relabel time for $n \leq 10{,}000$, but for the random grid network, the arc scans for relabels has a higher share for all n. With the increase in density of the random layered network, the dominance of the augmentation time over the relabel time increases. In the random grid network, arcs emanating from each node are

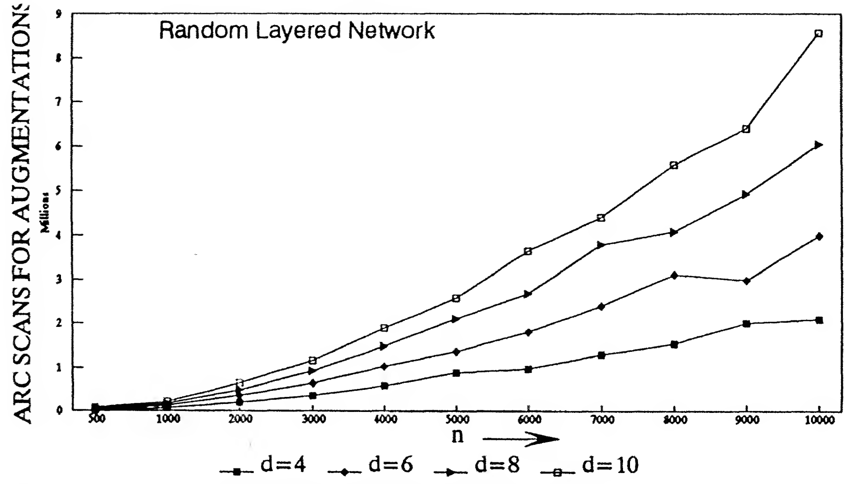


Figure 3.3. Arc scans for augmentations

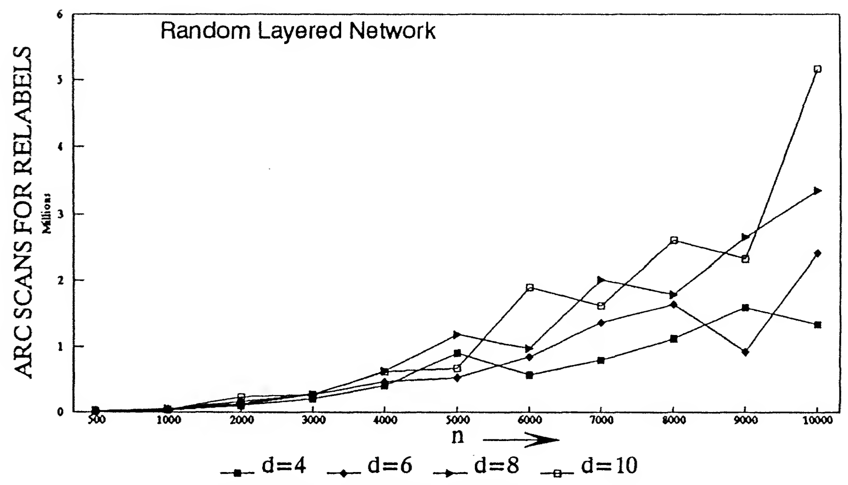


Figure 3.4. Arc scans for node relabels

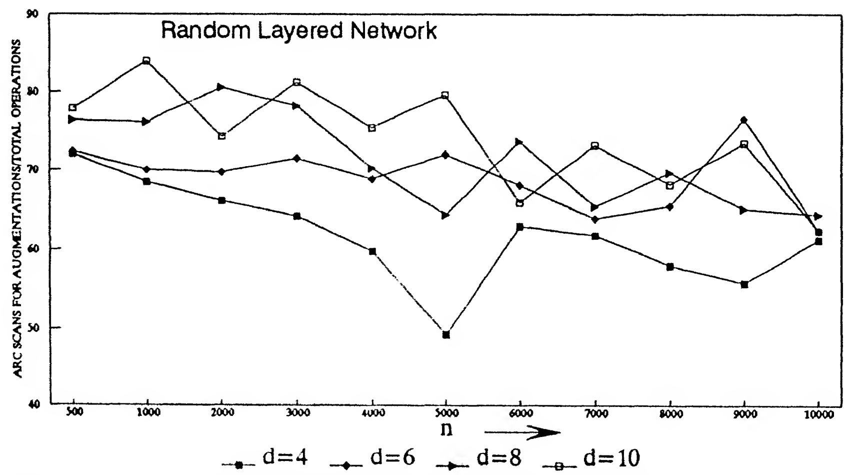


Figure 3.5. Identifying asymptotic bottleneck operation:
Ratio of arc scans for augmentations to total operations

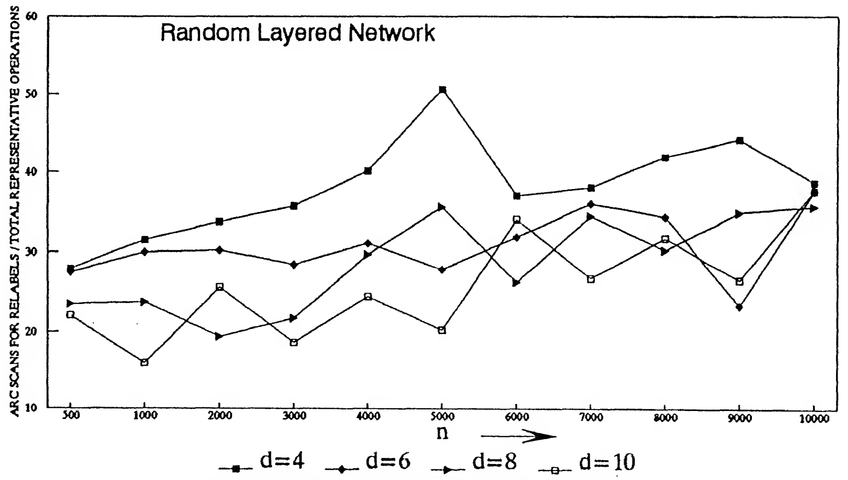


Figure 3.6 Identifying asymptotic bottleneck operation:
Ratio of arc scans for relabels to total operations

SHORTEST AUGMENTING PATH ALGORITHM

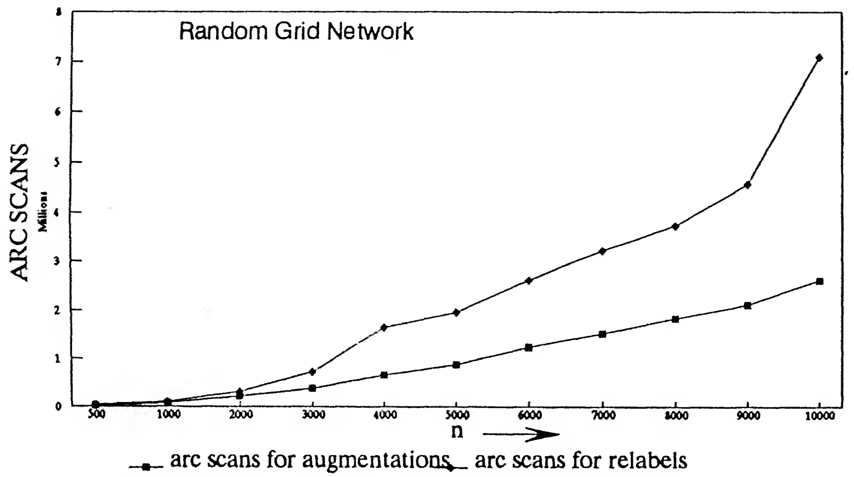


Figure 3.7. Represenatative operation counts: arc scans for augmentations and arc scans for relabels

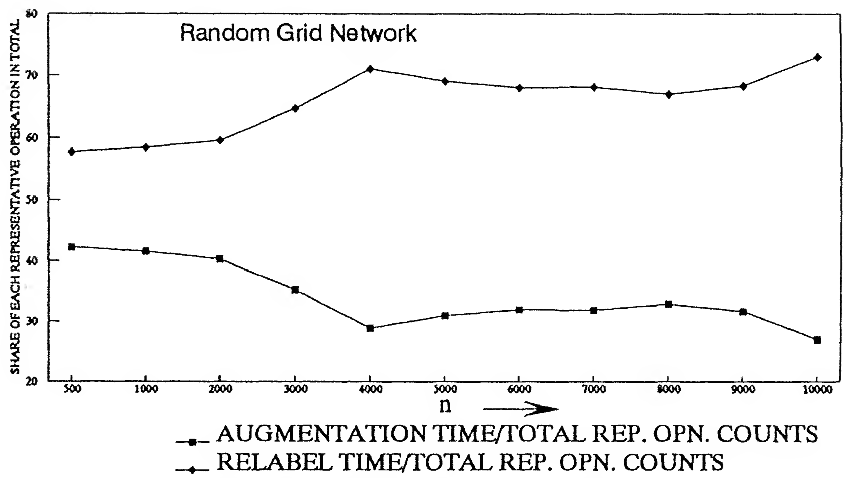


Figure 3.8. Identifying asymptotic bottleneck operation: augmentation time and relabel time

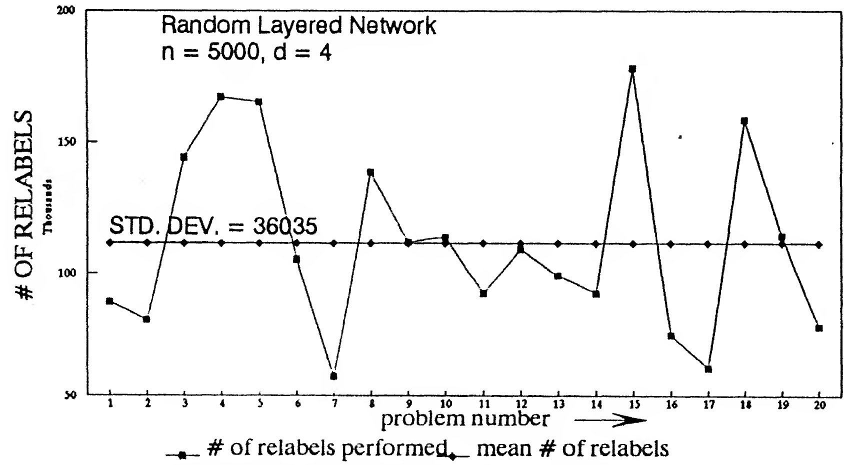


Figure 3.9. Variation of # of relabels for 20 problems solved for taking average values

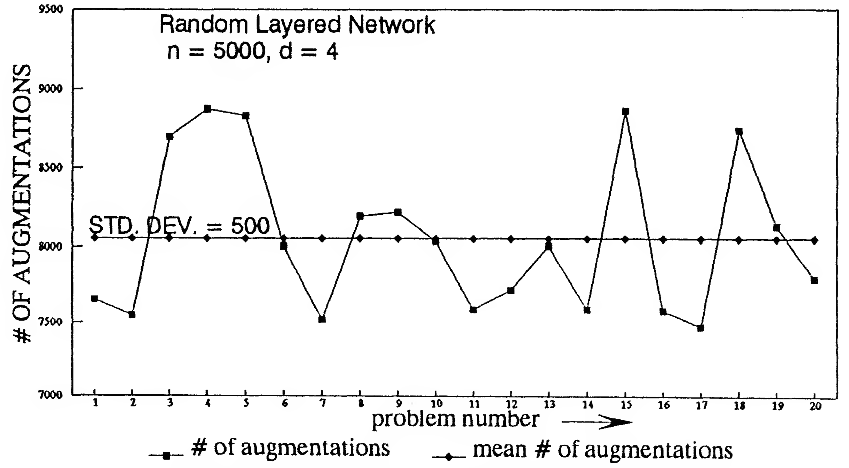


Figure 3.10. Variation of # of augmentations for 20 problems solved for taking average values

SHORTEST AUGMENTING PATH ALGORITHM

incident on two nodes in its own layer and on three nodes in the layer next to it. Since there are more alternate paths to the sink, saturation of some of the incident arcs to a node does not prevent it from inclusion in an augmenting path. This might explain the high number of relabels in the random grid network.

The erratic variation of arc scans for node relabels was a surprising observation. But this behavior of the arc scans for node relabels (and also for the number of relabels) was observed for all distance directed algorithms (for e.g. refer Figure 4.40 for relabels in highest distance version of preflow-push algorithms), for different networks (We also tested our codes on NETGEN by Klingman et al. [1974], purely random networks and Goldberg's network generator [1985]), for different input parameters for the network generators (the length to width ratio in random layered network and random grid network), for different initial seeds for the network generator, and also after taking the average for a large number of problems (30-40) for the same n. The arc scans for constructing layered networks in Dinic's algorithm too varied erratically (in fact, similar as the arc scans for node relabels for the shortest augmenting path algorithm). So we accept this behavior of the arc scans for relabels as influenced by network topology - some problems are hard and lead to more relabels. The variation of the total number of node relabels for the 20 distinct problems solved for each n and d was also remarkable. Figure 3.9 shows this variation for a representative problem. The similar variations in the arc scans for relabels and arc scans for augmentations [compare Figures 3.9 and 3.10] lead us to conclude that some problems are hard and both the arc scans for relabels and the arc scans for augmentations increase.

The virtual running time, a linear estimate of the CPU time for an instance I, for the shortest augmenting path algorithm is estimated using the arc scans for augmentations and the arc scans for relabels, in the following manner:

$$V(I) = C_A \text{ (arc scans for augmentations)} + C_R \text{(arc scans for relabels)},$$

where C_A and C_R are constants. We use multiple regression analysis to estimate C_A and C_R for networks of the same density but different sizes.

Network	Density (d)	C_A (10^{-7})	C_R (10^{-7})
Random Layered Network	4	78	69
	6	76	67
	8	74	70
	10	77	65
Random Grid Network		77	66

Table 3.1. Regression coefficients in virtual running time estimation for the shortest augmenting path algorithm

A plot of V(I)/CPU(I) in Figure 3.11 for different n shows that the virtual running time is a fairly good approximation of the CPU time. The error is less than 4% for all except 3 out of 45 cases and for $n \geq 5{,}000$, the error is less than 1%. This close approximation also confirms that the arc scans for augmentations and the arc scans for relabels are the representative operations for the shortest augmenting path algorithm.

We used the virtual time analysis to assess the proportion of time that the shortest augmenting path algorithm spends on the representative operations. We express the virtual time for the shortest augmenting path algorithm, as a function of the representative operations, for each density of the random layered network and for the grid network. For example, for the random layered network with density, d = 4, we express the virtual time as,

$$V(I) = 1.87(\text{arc scans for augmentations}) + 1.67(\text{arc scans for relabels})/239{,}000$$

We plot 1.87(arc scans for augmentations)/(1.87(arc scans for augmentations) + 1.67(arc scans for relabels)) and 1.67(arc scans for relabels)/(1.87(arc scans for augmentations) + 1.67(arc scans for relabels)) for different n and d. From Figures 3.12 and 3.13, we observe that though relabeling is the asymptotic bottleneck operation for the shortest augmenting path algorithm, it takes only 20%-40% of the time for the random layered network for $n \leq 10{,}000$. Hence the augmentation time is the bottleneck operation for random layered network of size upto 10,000 nodes. For the random grid network, the relabeling time is 55%-70%. Thus, for this network, relabeling is the bottleneck operation for even network of small sizes.

Figure 3.14 shows the CPU time taken by the shortest path algorithm for the different test problems.

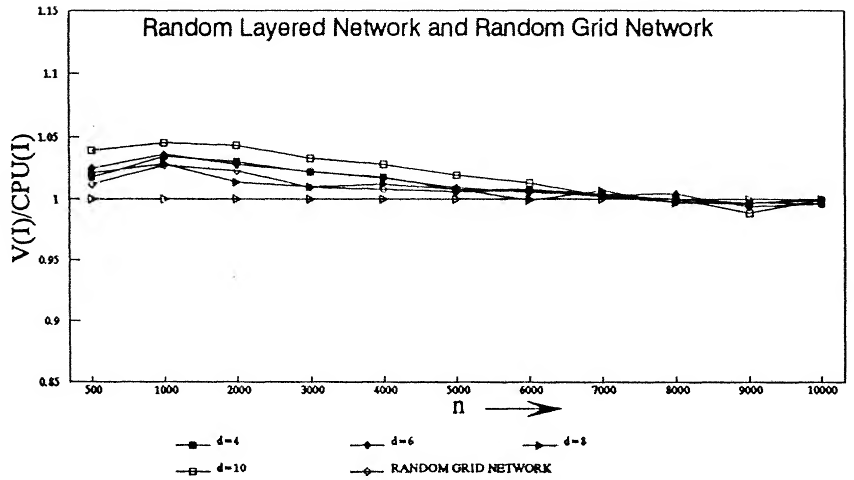


Figure 3.11. Virtual CPU time approximates CPU time

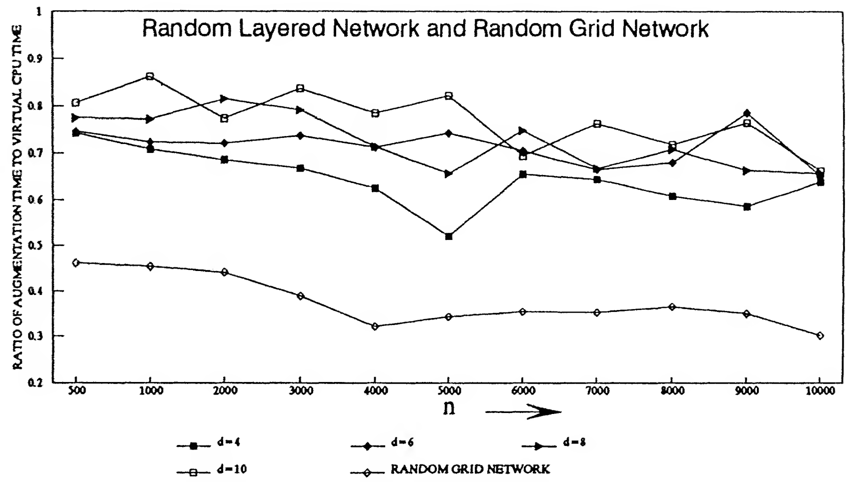


Figure 3.12. Percentage of virtual running time accounted by augmentation time

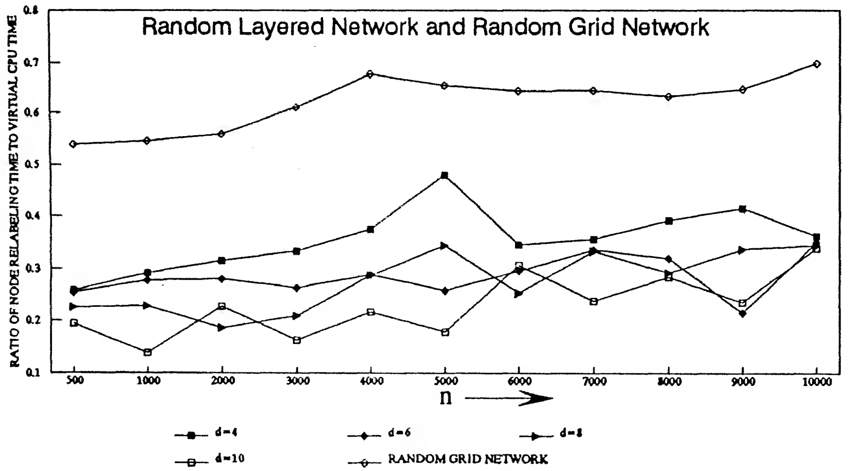


Figure 3.13. Percentage of virtual running time accounted by relabeling time

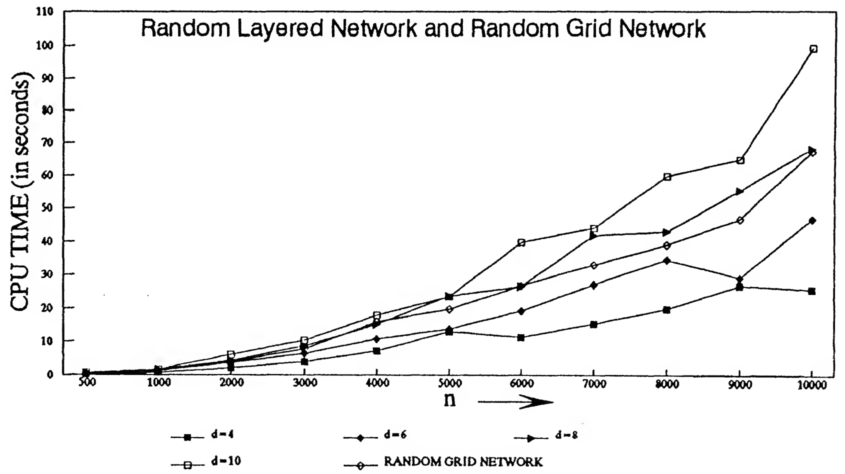


Figure 3.14. CPU time taken

SHORTEST AUGMENTING PATH ALGORITHM

We observed that about 85% of the maximum possible flow into the sink was sent to the sink using only 3 to 18% of the total node relabels [refer to Figure 3.15]. Figure 3.16 shows the relabels required to send flow into the sink for a large network. We can see that to send 85% of the total flow to the sink requires only 3% of the total node relabels. This prompted us to devise methods to reduce the relabels to improve the empirical running time. We updated the distance labels to their exact values at periodic intervals by a backward breadth-first search. This does not affect the worst-case complexity of the algorithm. This interval was decided by different methods. We updated distance labels in a variety of ways (for some constant α)

(a) Find exact distance labels, after every αn relabels;

(b) If between successive augmentations, if the number of relabels increases by αn, update the distance labels to their exact values;

(c) If after finding the exact distance labels, any node is relabeled more than α times, update the distance labels;

(d) Assign a distance label of n to any node which is relabeled more than α times to eliminate it from being a part of any augmenting path. When the source node is relabeled α times, we find the exact distance labels and again allow all nodes to be part of any augmenting path.

None of the above methods succeeded in improving the empirical running time. It was observed that finding the exact distance labels many times reduced the number of relabels only marginally, while increasing the CPU time taken by a larger amount.

We also used a two phase approach to send the flow to the sink by switching to the labeling algorithm, at an appropriate time, to send the flow to the sink in the last stages when a large number of relabelings take place. Labeling algorithm is much slower than the shortest augmenting path algorithm and the running time worsened.

A similar study on the number of augmentations showed that approximately the same percentage of the maximum possible flow in a network was sent to the sink as the percentage of total necessary augmentations that sent this flow [see Figure 3.17].

The use of the *numb* array, described in Section 3.2, improves the empirical running time remarkably by reducing the relabels. Table 3.2 is a fair representative of the power of the *numb* array.

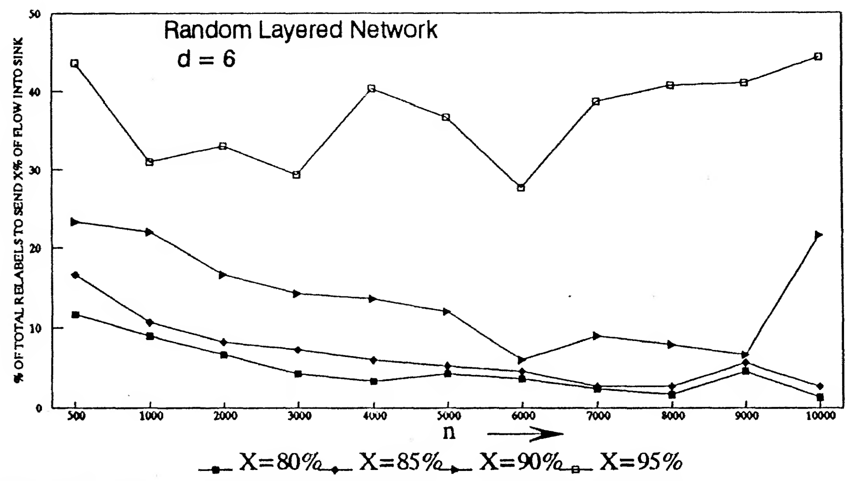


Figure 3.15. Flow into sink as a function of relabels: Percentage of relabels required to send X% of flow into sink

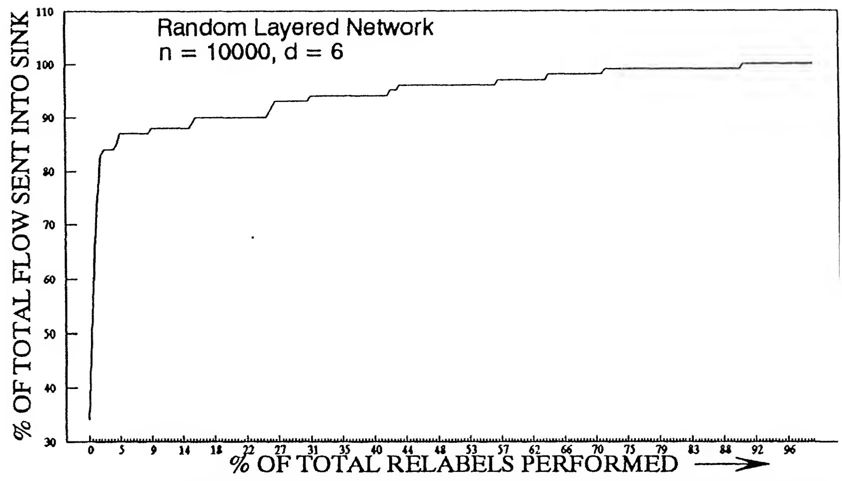


Figure 3.16. Percentage of relabels performed to send flow into sink for a large network

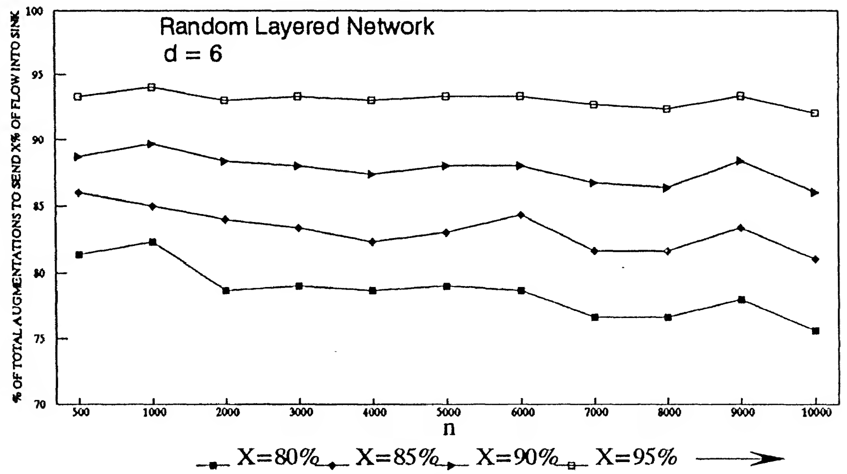


Figure 3.17. Flow into sink as a function of augmentations: Percentage of augmentations required to send X% of flow into sink

	# OF RELABELS		CPU TIME (in seconds)	
n	with *numb* array	without *numb* array	with *numb* array	without *numb* array
500	1380	47282	0.41	3.67
1000	4343	294900	1.19	22.69
2000	12816	1044614	3.88	81.17
3000	21102	2054433	6.46	162.18

Table 3.2. Use of *numb* array to speed up the shortest augmenting path algorithm.

The use of the **second** array, described in Section 3.2, to reduce the arc scans during the relabel operation succeeds in reducing the arc scans by upto 20-40% while bringing down the CPU time by only 5-10%. This also shows that the relabel time for network sizes upto 10,000 nodes and 100,000 arcs is not significantly large. We did not include this modification in our code for the shortest augmenting path algorithm while performing other tests.

We studied the growth rate of the bottleneck operations as a function of the problem size n. We used the functional form, operation count = n^{γ} and plotted γ = log(operation count)/log(n). The shortest augmenting path algorithm performs at most nm/2 augmentations and the total number of relabels is at most n^2. We observe that the augmentations vary between $n^{1.04}$ and $n^{1.25}$ [see Figure 3.18] and the number of relabel operations vary between $n^{1.1}$ and $n^{1.3}$ for the random layered network and between $n^{1.3}$ and $n^{1.5}$ for the random grid network [see Figure 3.19]. For the shortest augmenting path algorithm the augmentation time is $O(n^2m)$ and relabeling time is O(nm). We observe that the augmentation time is between $n^{1.6}$ and $n^{1.8}$ [refer to Figure 3.20] and the relabel time is between $n^{1.45}$ and $n^{1.7}$ [refer to Figure 3.21].

3.6 COMPUTATIONAL RESULTS FOR DINIC'S ALGORITHM

The investigations on the Dinic's algorithm confirm that the shortest augmenting path algorithm is similar to the Dinic's algorithm, except that the former uses residual network instead of Dinic's layered network. We noted in Section 3.3 that the arc scans for augmentations and arc scans for constructing layered networks are the representative operations for the Dinic's algorithm. Some highlights of our testing of Dinic's algorithm are

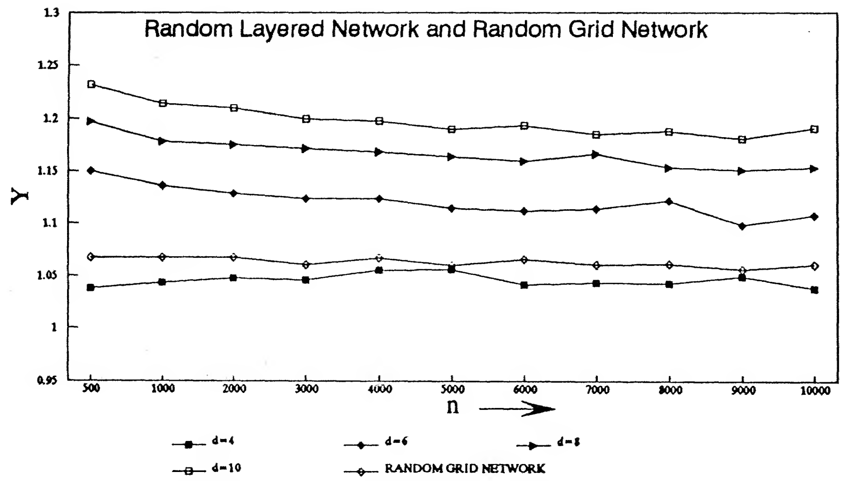


Figure 3.18. Growth rate of augmentations
Y = Log (# of augmentations) / Log (n)

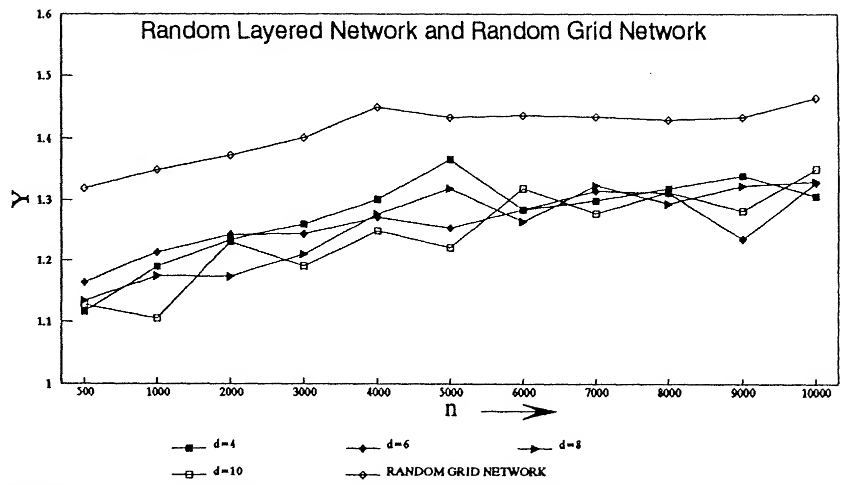


Figure 3.19. Growth rate of node relabels
Y = Log (# of relabels) / Log(n)

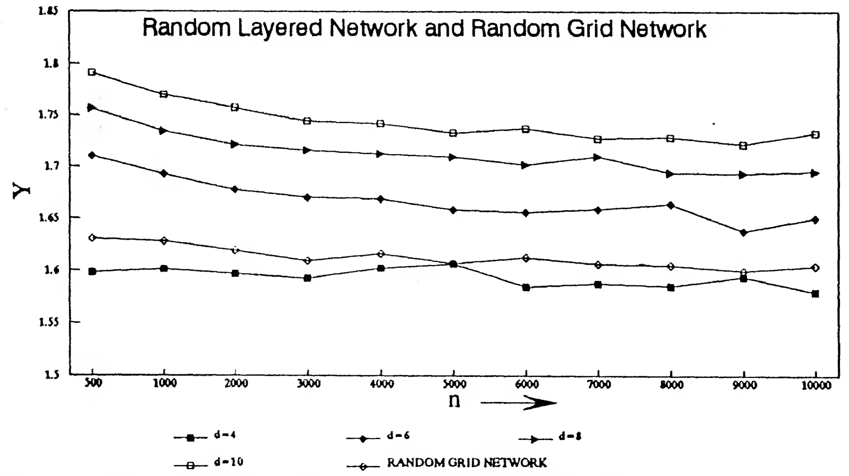


Figure 3.20. Growth rate of augmentation time
Y = Log (arc scans for augmentations) / Log (n)

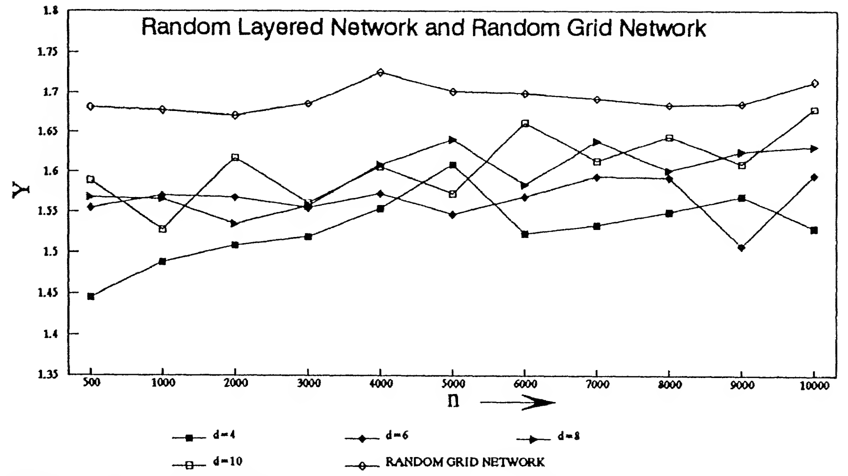


Figure 3.21. Growth rate of relabeling time
Y = Log (arc scans for node relabels) / Log (n)

(i) The arc scans for constructing layered networks is the asymptotic bottleneck operation for the Dinic's algorithm, as evident from Figures 3.24 and 3.25.

(ii) The arc scans for augmentations performed by Dinic's algorithm is almost equal to those performed by the shortest augmenting path algorithm [compare Figures 3.22 and 3.3, and Figures 3.26 and 3.7]. Similarly, the arc scans for constructing layered networks is almost equal to the arc scans for relabels in the shortest augmenting path algorithm [compare Figures 3.23 and 3.4, and Figures 3.26 and 3.7].

(iii) Comparing Figures 3.24, 3.25, 3.27 with Figures 3.5, 3.6, 3.8, respectively, we can see that for the Dinic's algorithm and the shortest augmenting path algorithm, the ratio of their two representative operation counts are similar. The erratic variation of the arc scans for layered network can be observed in these figures.

(iv) Figure 3.28 shows that Dinic's algorithm constructs many more [20%-30%] layered networks for the random grid network than for the random layered networks.

(v) We can observe in Figure 3.29 that only 2 layered networks are required to send 85% of flow into the sink for a large network of 10,000 nodes and 38 subsequent layered networks are used to send the remaining 15% flow into the sink.

(vi) The virtual running time for the Dinic's algorithm is estimated as,

$$V(I) = C_A \text{ (arc scans for augmentations) } + C_L \text{(arc scans for constructing layered networks)},$$

where C_A and C_L are constants. We use multiple regression analysis to estimate C_A and C_L for networks of the same density but different sizes.

Network	Density (d)	C_A (10^{-7})	C_L (10^{-7})
Random Layered Network	4	85	75
	6	81	78
	8	77	86
	10	80	79
Random Grid Network		81	69

Table 3.3 Regression coefficients in virtual running time estimation for Dinic's algorithm.

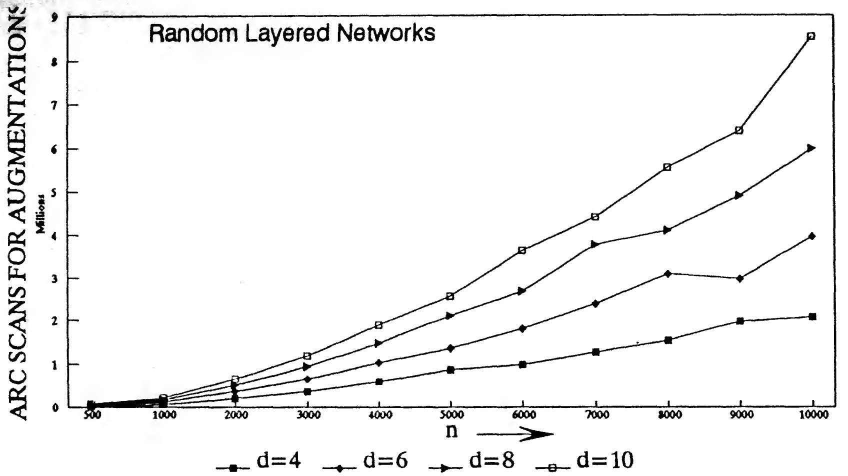


Figure 3.22. Arc scans for augmentations

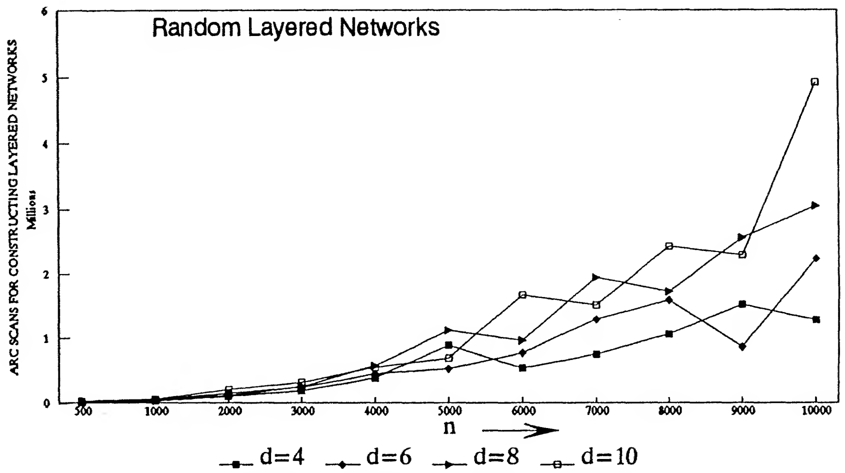


Figure 3.23. Arc scans for constructing layered networks

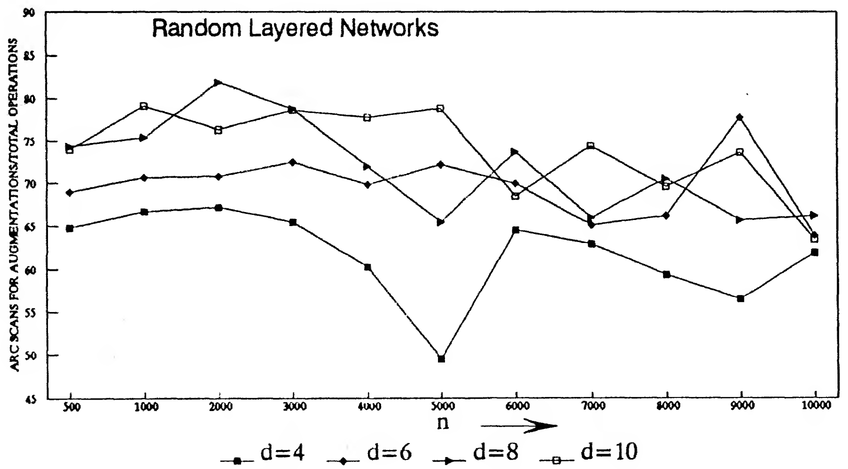


Figure 3.24. Identifying asymptotic bottleneck operation: Ratio of arc scans for augmentations to total operations

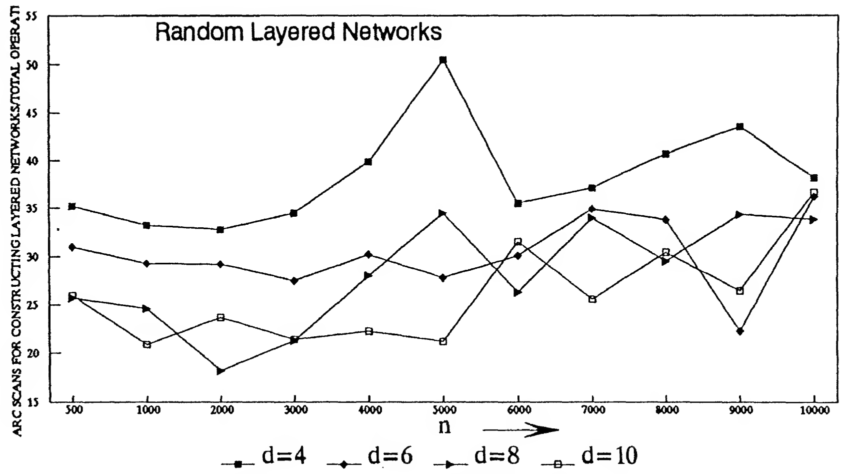


Figure 3.25. Identifying asymptotic bottleneck operation: Ratio of arc scans for constructing layered networks to total operations

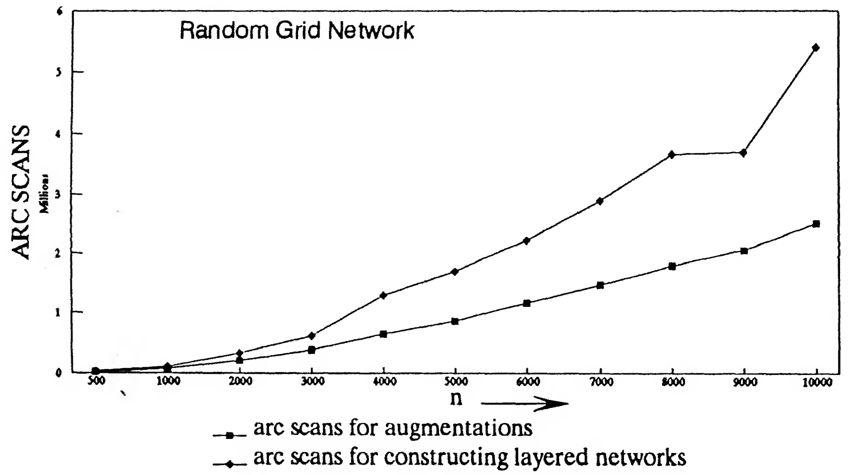


Figure 3.26. Representative Operations: arc scans for augmentations and arc scans for constructing layered networks

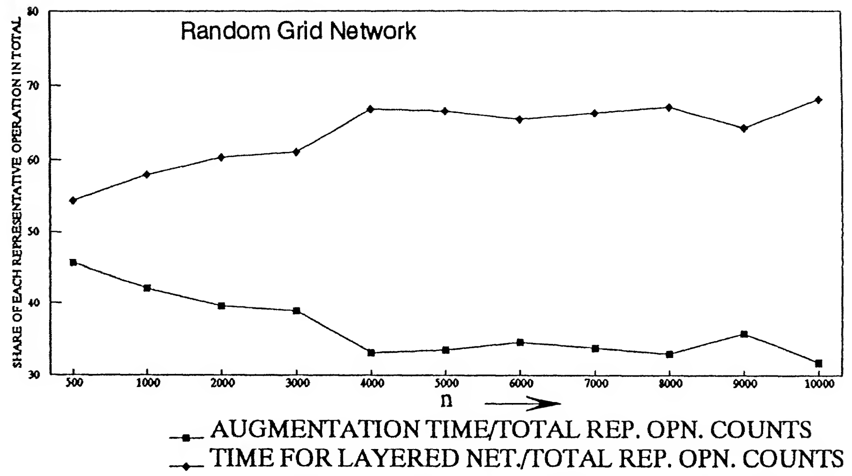


Figure 3.27. Identifying asymptotic bottleneck operation: augmentation time and relabel time

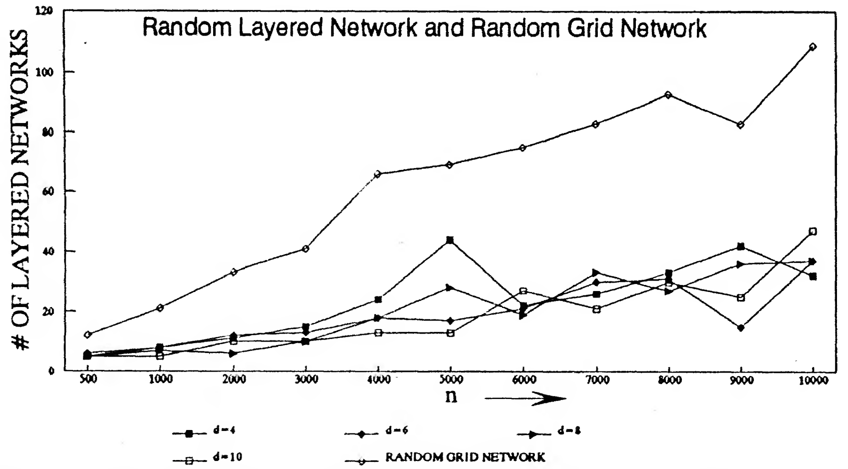


Figure 3.28. Number of layered networks constructed

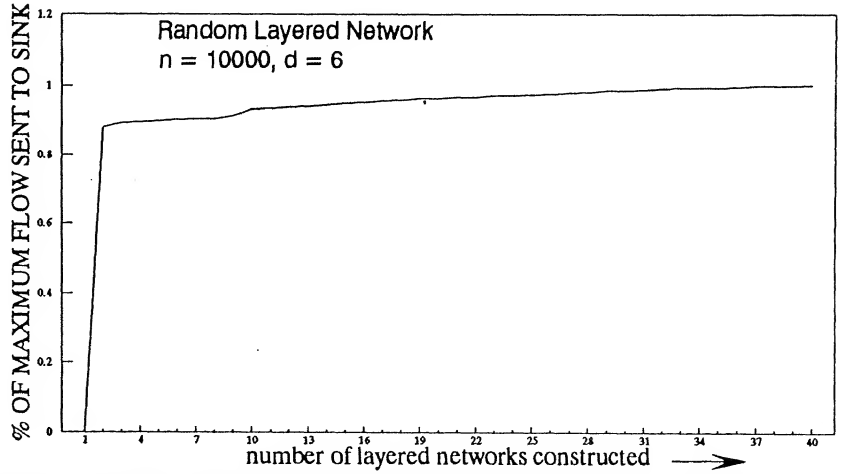


Figure 3.29. Flow sent into sink during each layered network

DINIC'S ALGORITHM

The virtual running time is a fair approximation of the CPU time, as is evident from Figure 3.30.

(vii) From Figures 3.31 and 3.32, it is clear that for the random layered network the time for constructing layered networks is not a bottleneck operation for networks of sizes upto $n \leq 10{,}000$. It is the augmentation time which is dominant for such random layered networks. But, the time for constructing layered networks is dominant for all sizes of the random grid network.

(viii) Comparing Figures 3.33 and 3.14, we observe that both the Dinic's algorithm and shortest augmenting path algorithm take almost the same CPU time to solve the same problem instances.

(ix) The augmentations vary between $n^{1.05}$ and $n^{1.25}$ [see Figure 3.34] and the augmentation time is between $n^{1.6}$ and $n^{1.8}$ [see Figure 3.36]. The growth rate of the number of layered networks constructed is around $n^{0.25}$ and $n^{0.4}$ for the random layered network and around $n^{0.4}$ to $n^{0.5}$ for the random grid network [refer to Figure 3.35]. The time for constructing layered networks varies between $n^{1.5}$ and $n^{1.7}$ for the two networks [see Figure 3.37].

3.7 COMPUTATIONAL RESULTS FOR CAPACITY SCALING ALGORITHM

We implemented Gabow's capacity scaling algorithm using the shortest paths to send the flow from the source to the sink. We noted in Section 3.4 that the arc scans for the augmentations and the arc scans for relabels are the representative operations for the capacity scaling algorithm.

The slope of the curves for the arc scans for augmentations and the arc scans for node relabels is similar to those for the shortest augmenting path algorithm. For the random layered network, the arc scans for augmentations show a fairly smooth increasing trend but the arc scans for relabels vary erratically while, in general, increasing with the network size [see Figures 3.38 and 3.39]. The increase in the arc scans for relabels is fairly uniform with the increase in the size of the random grid network [see, Figure 3.43]. The arc scans for augmentations increase smoothly for the random grid network [see, Figure 3.42].

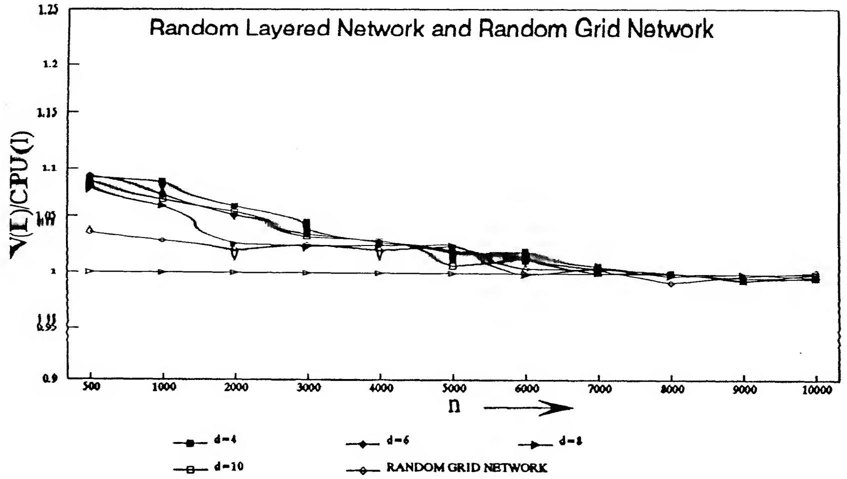


Figure 3.30. Determining how well virtual running time the CPU time

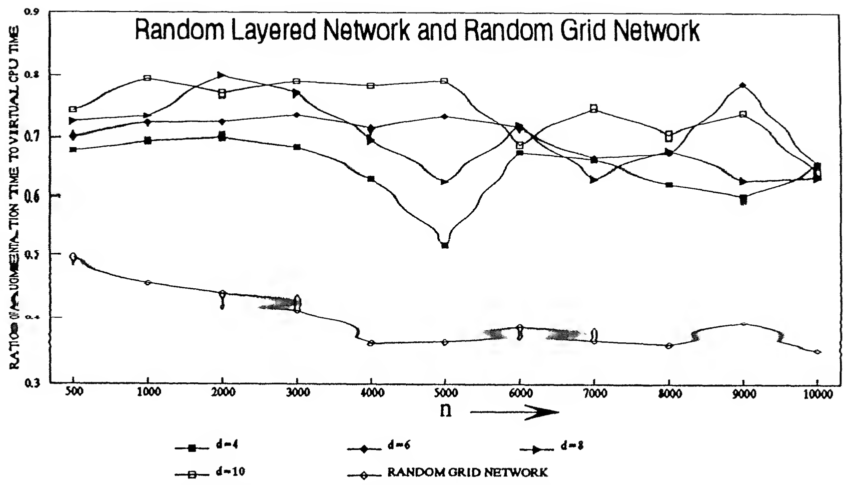


Figure 3.31. Percentage of virtual running time accounted by augmentation time

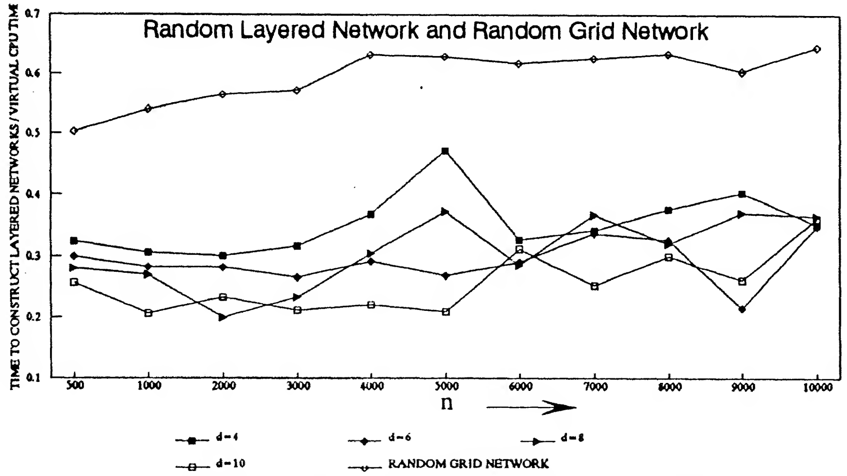


Figure 3.32. Percentage of running time accounted by time to construct layered networks

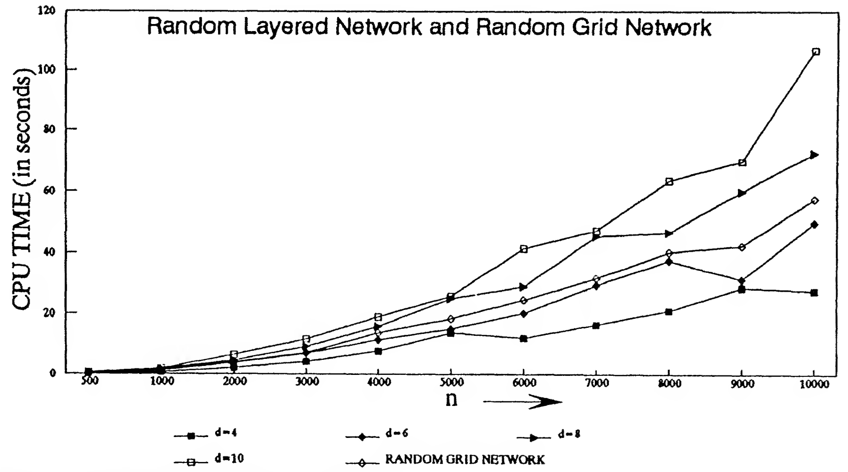


Figure 3.33. CPU time taken

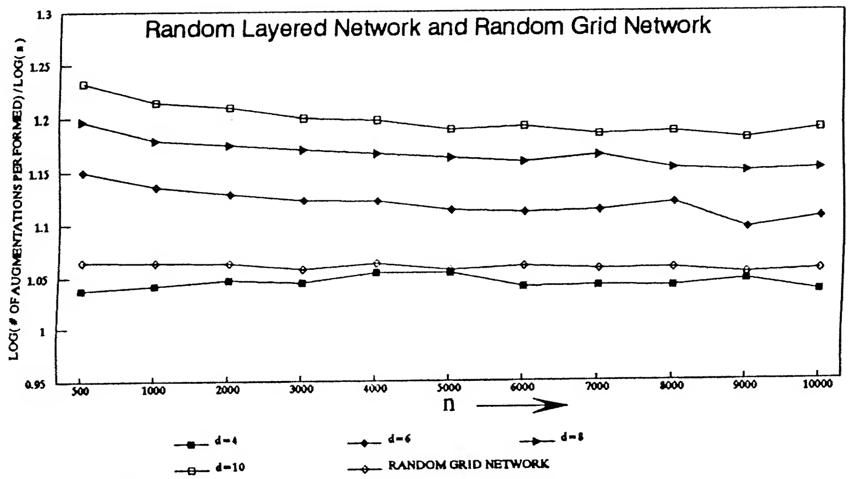


Figure 3.34. Growth rate of augmentations

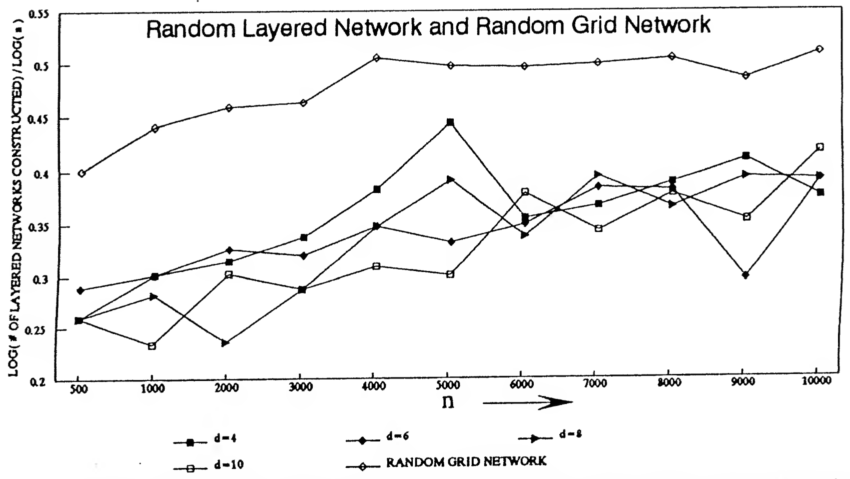


Figure 3.35. Growth rate of # of layered networks constructed

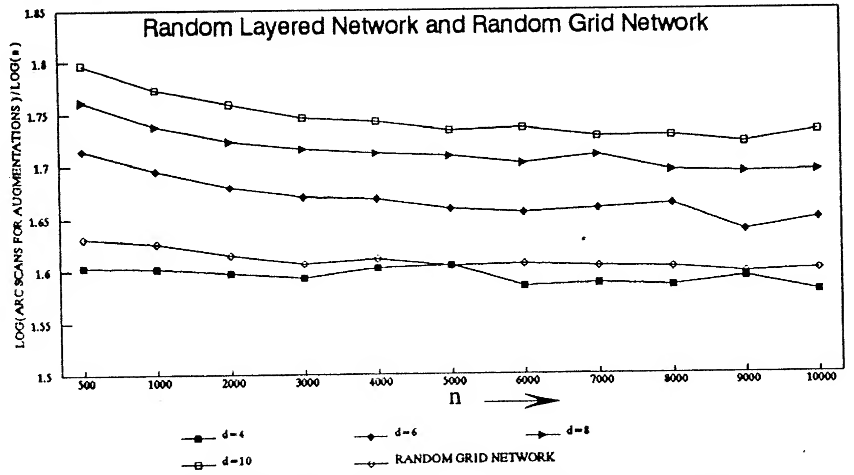


Figure 3.36. Growth rate of arc scans for augmentations

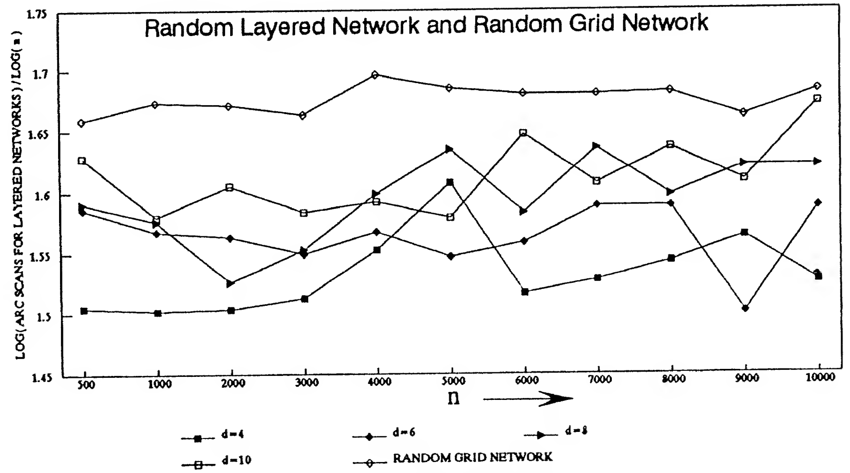


Figure 3.37. Growth rate of arc scans for constructing layered networks

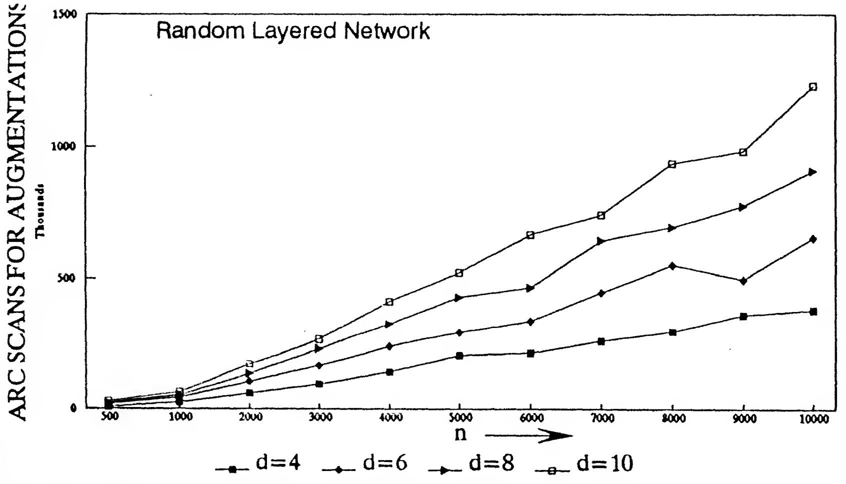


Figure 3.38. Representative Operation: arc scans for augmentations.

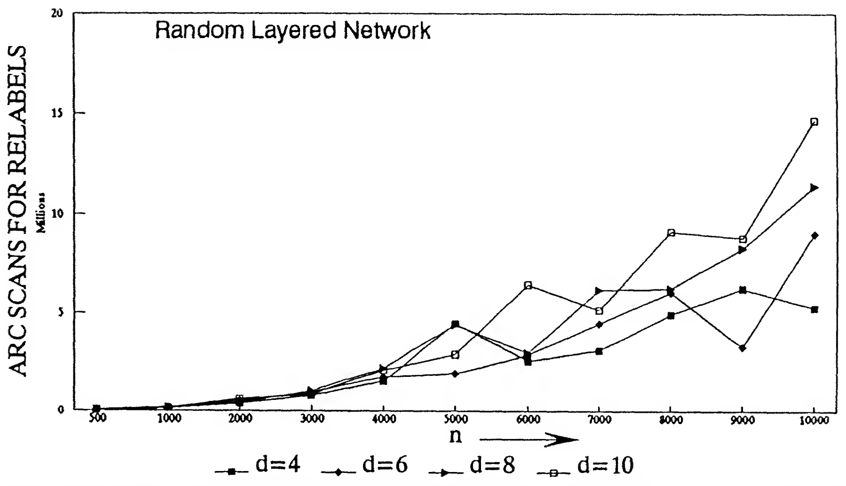


Figure 3.39. Representative Operation: arc scans for relabels

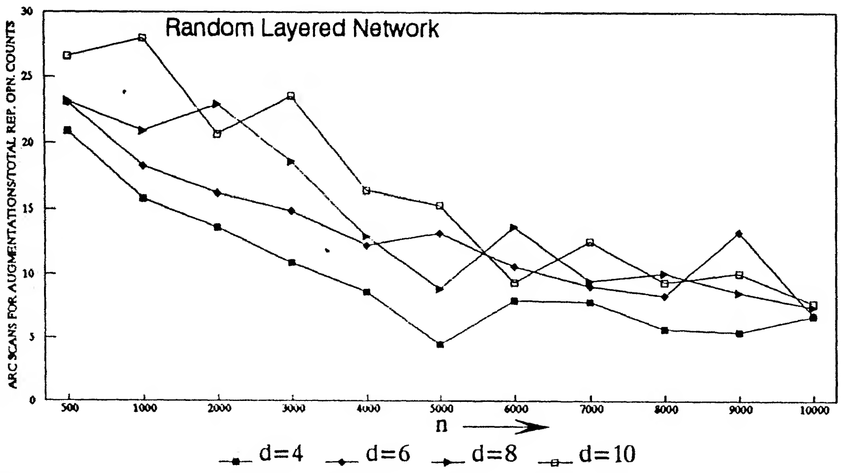


Figure 3.40. Identifying asymptotic bottleneck operation: ratio of arc scans for augmentations to total rep. opn. count

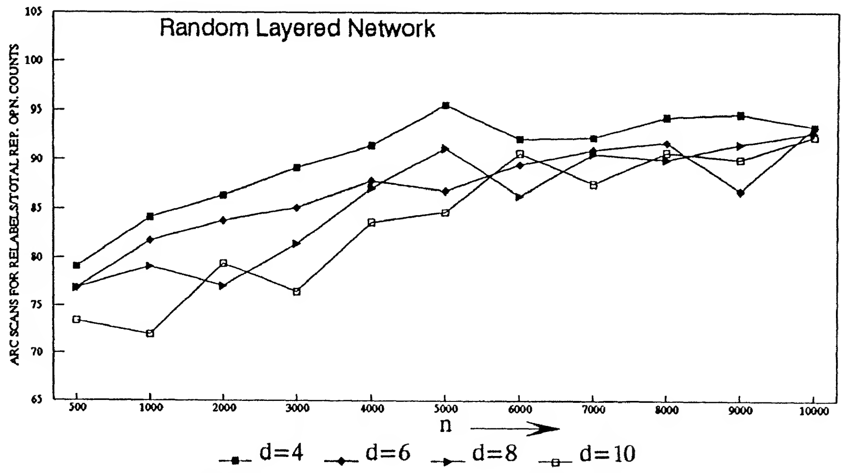


Figure 3.41. Identifying asymptotic bottleneck operation: ratio of arc scans for relabels to total rep. opn. count

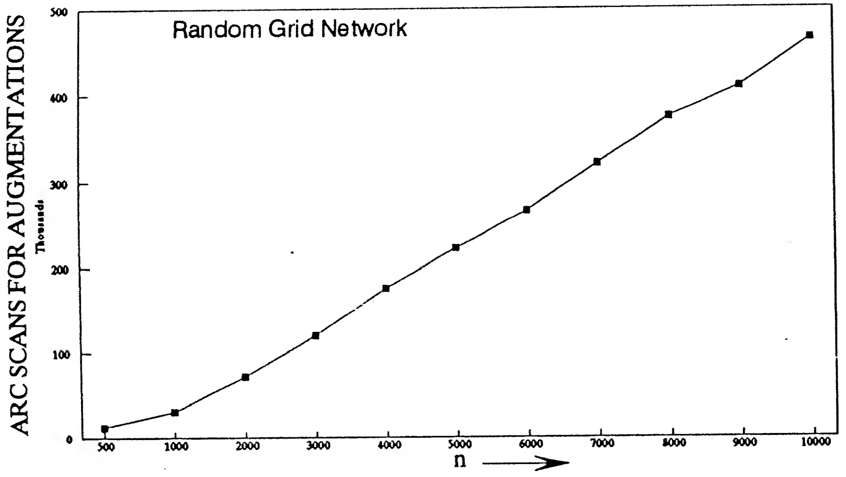


Figure 3.42. Representative operation: arc scans for augmentations

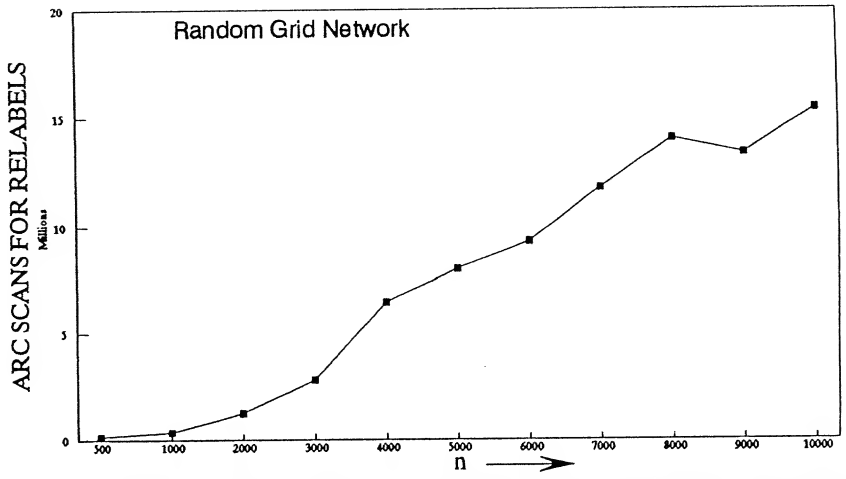


Figure 3.43. Representative operation: arc scans for relabels

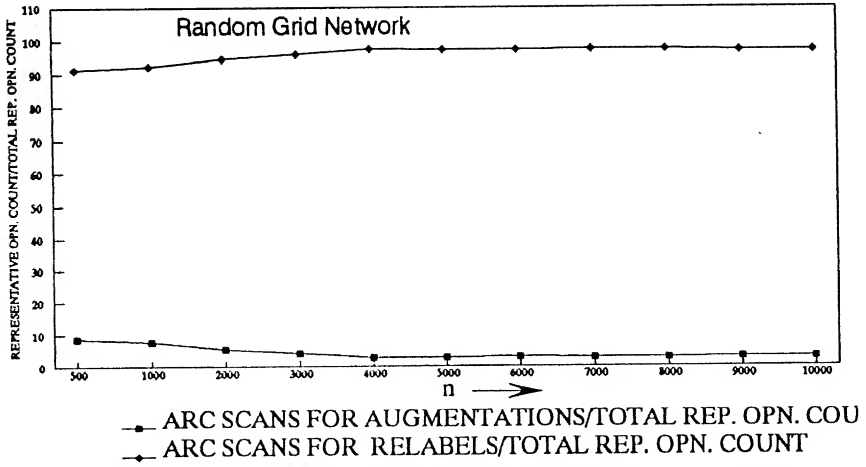


Figure. 3.44. Identifying asymptotic bottleneck operation: ratio of rep. opn. count to total rep. opn. count

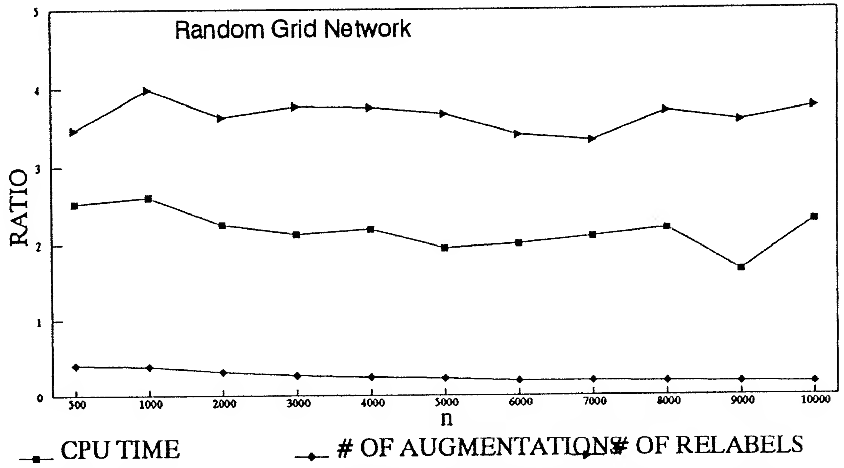


Figure 3.45. Ratio of count for capacity scaling algorithm to that for shortest augmenting path algorithm

CAPACITY SCALING ALGORITHM

Plotting the ratio of arc scans for node relabels to the sum of the arc scans for relabels and the arc scans for augmentations, we observe that for both the networks the curves show an increasing trend and the arc scans for relabels have a rising and dominant share in the sum (70%-95% for random layered network and 90%-95% for random grid network) for all network sizes [refer to Figure 3.40, 3.41, and 3.44]. Hence, the arc scans for node relabels is the asymptotic bottleneck operation for the capacity scaling algorithm.

The large number of relabels done to send the maximal flow in each Δ–scaling phase results in a large number of node relabels to send the maximum flow to the sink. The reduction in the augmentation time is at a high cost for the relabel time. We observed that the capacity scaling algorithm performs almost 3-4 times the relabels done by the shortest augmenting path algorithm while reducing the augmentations by more than half. The CPU time taken by the capacity scaling algorithm is about 2-2.5 times that taken by the shortest augmenting path algorithm. Figure 3.45 shows this comparison for a representative network.

We varied the scaling factor to observe its affect on the augmentation time, relabel time and the CPU time. As the scaling factor increases, the capacity scaling algorithm tends towards solving essentially shortest augmenting path algorithms. Hence the relabel time and the CPU time increase and the augmentation time decreases with increasing Δ. As seen in Figures 3.46, 3.47 and 3.48, these values tend towards the values for the shortest augmenting path algorithm with increasing scaling factor.

We can observe in Figure 3.49 that the CPU time taken by this code for the random layered network of densities upto 10 is lower than the CPU time taken by the random grid network. This is different from the empirical performance by the shortest augmenting path algorithm where the time taken on the random grid network is lesser than the time taken by the algorithm for random layered networks of densities of 8 and 10. We can compare Figures 3.7 and 3.43 and see that the number of arc scans for relabeling for the capacity scaling algorithm is about 3 to 5 times that for shortest augmenting path algorithm.

We estimated the virtual running time for an instance I for the capacity scaling algorithm using the expression:

$V(I) = C_A$ (arc scans for augmentations) + C_R(arc scans for relabels),

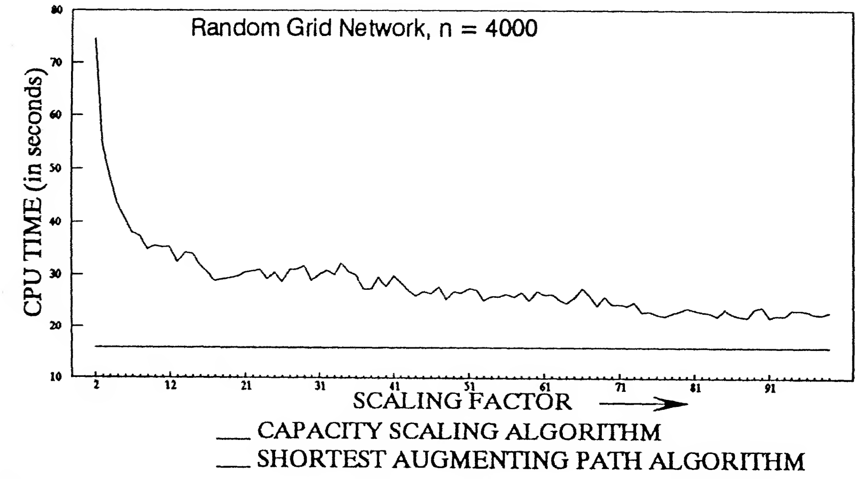


Figure 3.46. Effect of varying scaling factor on CPU time and approximation to corresponding value for shortest augmenting path algorithm

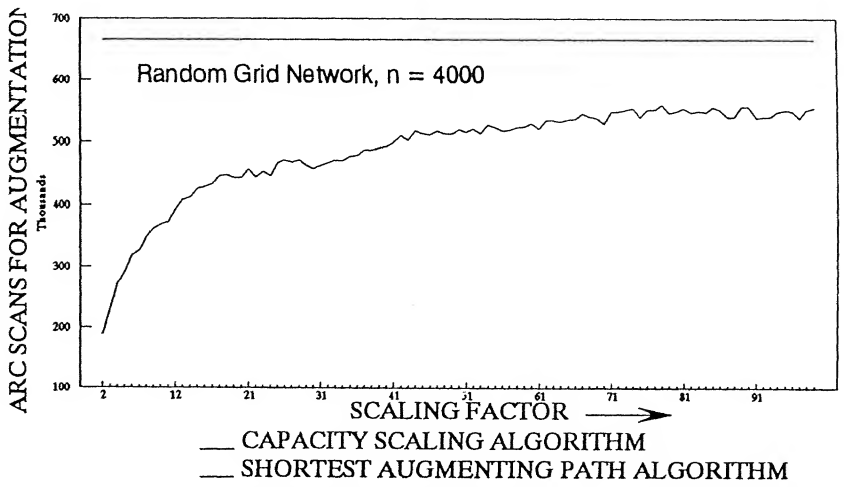


Figure 3.47. Effect of varying scaling factor on augmentation tir and approximation to corresponding value for shortest augme path algorithm

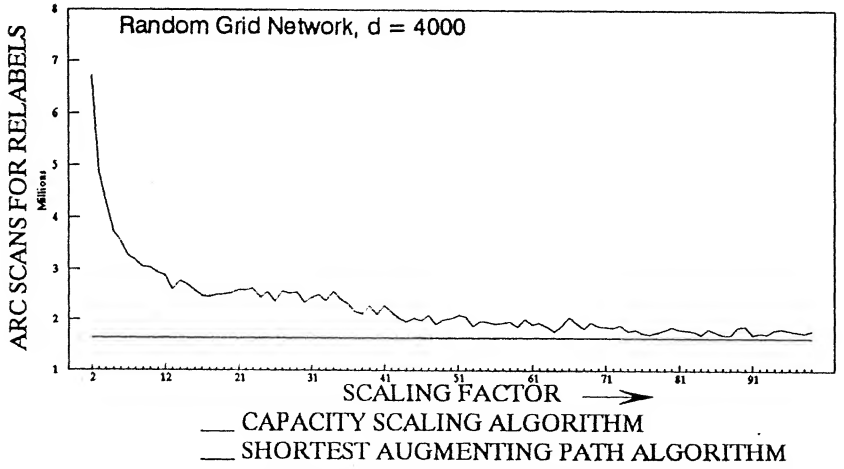


Figure 3.48. Effect of varying scaling factor on relabeling time and approximation to corresponding value for shortest augmenting

CAPACITY SCALING ALGORITHM

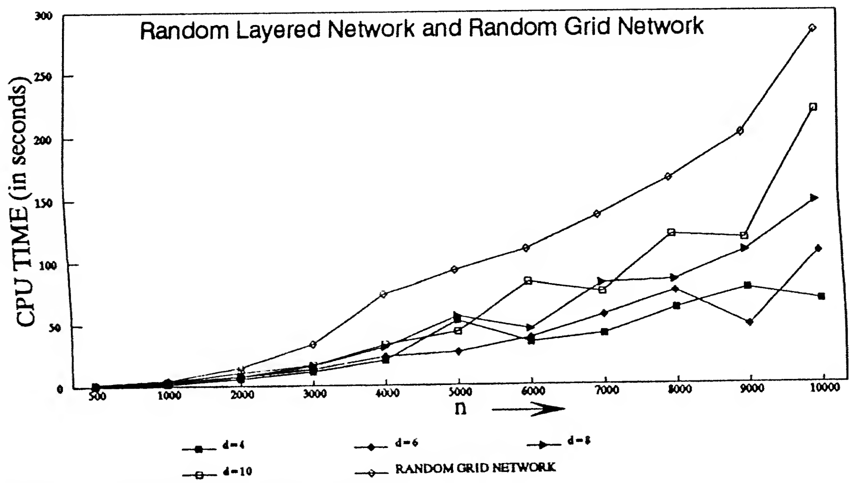


Figure 3.49. CPU time taken

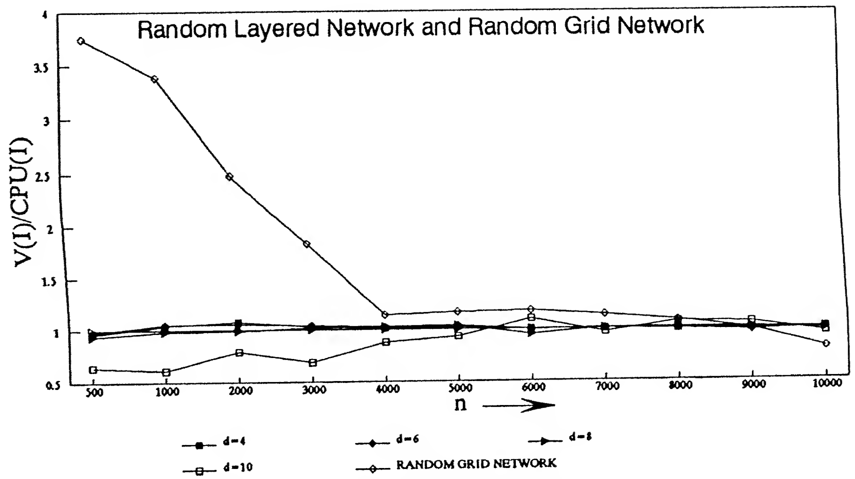


Figure 3.50. Estimation of CPU time by virtual running time

CAPACITY SCALING ALGORITHM

where C_A and C_R are constants. We use multiple regression analysis to estimate C_A and C_R for networks of the same density but different sizes.

Network	Density (d)	C_A (10^{-6})	C_R (10^{-6})
Random Layered Network	4	35	10
	6	29	9.8
	8	21	11
	10	-5.5	14
Random Grid Network		567	-2.2

Table 3.4. Regression coefficients in virtual running time estimation for the capacity scaling algorithm.

Figure 3.50 shows that the virtual running time is a poor estimate of the CPU time for network of size $n \leq 4{,}000$ but for larger networks, it approximates the CPU time fairly well.

CHAPTER 4

PREFLOW - PUSH ALGORITHMS

4.1 INTRODUCTION

The *preflow* approach has emerged as a powerful technique to solve maximum flow problems and obtain speed-ups, both theoretically and computationally, over the previously developed augmenting path algorithms. The preflow approach was introduced by Karzanov [1974]. In this approach, we relax the mass balance constraints at intermediate steps of the algorithm. Flow is sent along the shortest paths from the source to the sink but each flow is sent along individual arcs and not along augmenting paths. In this chapter, we discuss the classical Karzanov's algorithm and different implementations of the more recently developed Goldberg and Tarjan's preflow-push algorithms.

4.2 KARZANOV'S ALGORITHM

This algorithm uses layered networks and permits peflows [as defined in Section 1.5]. Flow is pushed along arcs and we try to achieve mass balance at all nodes except s and t. To reduce the excess at a node, we push flow towards nodes closer to sink.

We use some definitions different from those described in Section 1.5. A node i is called *unbalanced* if excess(i)>0 and *balanced* if excess(i)=0. Unbalanced nodes are also grouped as *inactive* and *active*. A node is called inactive if it remains unbalanced after an effort to push its excess to nodes closer to sink has been made after the latest time that it became unbalanced. We term it as active otherwise. The active nodes are stored in a queue and the inactive nodes in a stack. A stack is also maintained to store the history of the flow sent to a node - the arc used to send the flow and the amount of flow sent is stored in this stack.

This algorithm uses the following two subroutines to get rid of the excess on the unbalanced nodes.

(i) PUSH(i) chooses unsaturated arcs (i, j) in the layered network and increases flow, δ, on them until excess(i) becomes zero or all emanating arcs are saturated. Let arc number a = (i, j). It also pushes (a, δ) in the history stack for node i and adds node j to the active queue if its excess earlier was nil.

(ii) BALANCE(i) iteratively pops element (a, δ) from the history stack for node i a
reduces the flow on these entering arcs by δ until excess(i) = 0. After excess(i)=0,
deletes i and the arcs incident to it from the layered network.

The maximum flow is obtained by the following steps:

(i) excess(s) =∞, and, for all $i \in N'$-{s, t}, excess(i) = 0.

(ii) Until active queue is empty, delete node i from active queue and PUSH(
This way we choose nodes closer to the source node to execute PUSH(i). If excess(i) >
after PUSH(i), push i to inactive stack.

(iii) If inactive stack is empty, STOP; else pop a node i from the inactive stac
Let the distance of node i from the sink node be l'= l(i).

(iv) BALANCE(i) for all nodes in inactive stack with length from the sin
node, l(i)=l'. Thus we execute BALANCE(i) for all nodes in the layer nearest to th
sink. GO TO (ii).

Due to the order of choosing nodes for reducing the excess, it can be shown tha
each node is balanced at most once. Hence, a maximum flow can be found in $O(n^3)$ time
The active queue and the inactive stack are implemented using two arrays of size n
since the two sets are disjoint. In implementing the history stack for each node, we
observe that only the latest increment on each arc is to be recorded. So these stacks are
implemented by two arrays of size m for LIFO lists and an array of size n for head
pointers to them. To store the excess on each node, we use an array of size n and to store
the arcs emanating from a node in the layered network, we use an array of size m.
Hence, the total space used by Karzanov's algorithm is O(4n + 2m).

The non-saturating pushes performed by the algorithm and the arc scans for
constructing layered networks are the two representative operations for this
algorithm.

4.3 PREFLOW-PUSH ALGORITHMS

The preflow-push algorithms due to Goldberg and Tarjan [1986] are
conceptually similar to Karzanov's algorithm but use distance labels instead of
layered networks. These algorithms maintain a preflow and proceed by examining
active nodes (i.e., nodes with positive excess). The basic step in the algorithm is to
select an active node and to attempt to get rid of its excess by pushing flow on

consideration has no admissible arc then we increase its distance label so that it creates at least one admissible arc. The algorithm terminates when it has no active nodes. The preflow-push algorithm uses the following subroutines.

```
procedure pre-process;
begin
    $x: = 0$;
    compute the exact distance labels $d(i)$;
    $x_{sj}: = u_{sj}$ for each arc $(s, j) \in A(s)$;
    $d(s): = n$;
end;
```

```
procedure push/relabel(i);
begin
    if the network contains an admissible arc $(i, j)$ then
      push $\delta: = \min\{e(i), r_{ij}\}$ units of flow from node i to node j
    else replace $d(i)$ by $\min\{d(j)+1 : (i, j) \in A(i)$ and $r_{ij} > 0\}$;
end;
```

A push of δ units from node i to node j decreases both $e(i)$ and r_{ij} by δ units and increases both $e(j)$ and r_{ji} by δ units. We say that a push of δ units of flow on arc (i, j) is *saturating* if $\delta = r_{ij}$ and *non-saturating* otherwise. A non-saturating push at node i reduce $e(i)$ to zero. We refer to the process of increasing the distance label of a node as a *relabel* operation. The purpose of the relabel operation is to create at least one admissible arc on which the algorithm can perform further pushes.

The following generic version of the preflow-push algorithm combines the subroutines just described.

```
algorithm preflow-push;
begin
    pre-process;
    while the network contains an active node do
    begin
      select an active node i;
      push-relabel(i);
    end;
end;
```

In an iteration, the generic preflow-push algorithm selects a node say, node i, and performs a saturating push or a non-saturating push or relabels the node. If the algorithm performs a saturating push, then node i may still be active, but it is not mandatory for the algorithm to select this node again in the next iteration. The algorithm may select another node for *push/relabel*. However, it is easy to incorporate the rule that whenever the algorithm selects an active node, it keeps pushing flow from that node until either its excess becomes zero or it is relabeled. Consequently, there may be several saturating pushes followed by either a non-saturating push or a relabel operation. We associate this sequence of operations with a *node examination*. We shall henceforth assume that the preflow-push algorithms follow this rule.

In the *push/relabel(i)* step, we identify admissible arcs emanating from node i by using the same *current arc* data structure that we used in the shortest augmenting path algorithm. We have seen earlier that if each node is relabeled O(n) times then the total time spent in identifying admissible arcs is O(nm). The following additional results can be proved for the generic preflow-push algorithm.

Lemma 4.1. (a) The algorithm maintains valid distance labels at each step.

(b) Each relabel step increases the distance label of a node by at least one unit.

(c) During the intermediate steps of the algorithm, $d(i) \leq 2n$ for all $i \in N$.

(d) The algorithm performs at most nm saturating pushes.

Proof. Omitted. ■

Similar to the shortest augmenting path algorithm it can be easily shown that the generic preflow-push algorithm would take O(nm) time to perform saturating pushes, to relabel nodes, and to identify admissible arcs. The bottleneck step in the algorithm is the number of non-saturating pushes. It can be shown, using potential function arguments, that the generic version of the algorithm in which active nodes can be examined in any arbitrary order performs $O(n^2m)$ non-saturating pushes. Many specific rules for examining active nodes decrease the number of non-saturating pushes substantially. We consider the following four implementations.

Highest label preflow-push algorithm. This algorithm always pushes from an active node with the highest value of the distance label. It can be shown that this algorithm performs $O(n^2\sqrt{m})$ non-saturating pushes and runs in the same time. Our testing reveals this algorithm to be faster than all other algorithms we tested.

FIFO preflow-push algorithm. This algorithm examines the active node in the first-in first-out (FIFO) order. It can be shown that this algorithm runs in $O(n^3)$ time.

Wave algorithm. This algorithm is a hybrid version of the highest label and FIFO preflow-push algorithm. This algorithm also runs in $O(n^3)$ time.

Lowest label preflow-push algorithm. This algorithm always examines an active node with the smallest distance label. The complexity of this algorithm in $O(n^2m)$ which is same as that of the generic algorithm.

4.3(a) Highest label preflow-push algorithm

The highest label preflow-push algorithm always pushes flow from active node with the highest distance label. Let $h^* = \max\{d(i): i \text{ is active}\}$. The algorithm first examines nodes with distance label equal to h^* which push flow to nodes with distance label equal to h^*-1, and these nodes, in turn, push flow to nodes with distance label equal to h^*-2, and so on, until either a node is relabeled or no more active nodes are left. If a node is relabeled then h^* increases and the same process is repeated again. But if the algorithm does not relabel any node during n consecutive node examinations then all the excess reaches the sink (or the source) and the algorithm terminates. Since the algorithm performs at most $2n^2$ relabel operations, we immediately obtain a bound of $O(n^3)$ on the number of node examinations. As each node examination entails at most one non-saturating push, the highest label preflow-push algorithm performs $O(n^3)$ non-saturating pushes and runs in the same time. This bound of $O(n^3)$ for the algorithm was obtained by Goldberg and Tarjan [1988]. Cheriyan and Maheshwari [1989] later showed that the algorithm actually runs in $O(n^2\sqrt{m})$ time and that this bound is tight.

We now describe some implementation details of the highest label preflow-push algorithm. We first discuss how the algorithm selects an active node with highest distance label without too much effort. We use the following data structure to accomplish this. We maintain the lists, SLIST(k) = {i: i is active and d(i) = k}, for each k = 1,2, ... 2n-1 in the form of singly linked stacks, as described in Section 2.3. The index *NEXT(k)* points to the first node in *SLIST(k)* if *SLIST(k)* is non-empty and is zero otherwise. We define a variable *level* representing an upper bound on the highest value of k for which *SLIST(k)* is non-empty. In order to determine a node with highest distance label, we examine the lists *SLIST(level)*, *SLIST(level–1)*, ..., until we find a non-empty list, say *SLIST(p)*. We select any node in *SLIST(p)* and set *level* = p. Also, whenever the distance label of a node being examined increases, we set *level* equal to the new distance label of the node. Observe that the total increase in *level* is at most $2n^2$ (from Lemma 4.1(c)) hence the total decrease in *level* is at most $2n^2+n$.

Consequently, scanning the lists *SLIST(level), SLIST(level–1), ...,* in order to find the first non-empty list is not a bottleneck operations.

The highest label preflow-push algorithm terminates when all the excess is pushed to the sink or returns back to the source. This termination criteria is not attractive in practice because this results in too many relabels and too many pushes, a major portion of which is done after the algorithm has established a maximum flow. This inefficiency stems from the following fact. Suppose that an active node i becomes disconnected from the sink, i.e., has no path to sink in the residual network, and further suppose that $d(i) << n$. Since the excess of node i can not be sent to sink, it will eventually be pushed back to the source node. But for this to happen, $d(i) > d(s) = n$. Consequently, the distance label of node i will keep on increasing until it is sufficiently large so that the excess can be pushed to the source node. This is a very common phenomenon in preflow-push algorithms and needs to be avoided for them to become competitive with other maximum flow algorithms.

We overcame this drawback by implementing the two phase version of the preflow-push algorithms. This technique uses ideas contained in the papers of Goldberg and Tarjan [1986], Ahuja and Orlin [1989] and Derigs and Meier [1989]. We maintain the lists DLIST(k) = {1 : d(i) = k}; for each k = 1, 2, ..., n in the form of doubly linked lists, as described in Section 2.3. The set *DLIST(k)* consists of all nodes with distance label equal to k, and the index *FIRST(k)* points to the first node in *DLIST(k)* if *DLIST(k)* is non-empty and is zero otherwise. We initialize these lists when initial distance labels are computed by the breadth-first-search. Subsequently, we update these lists whenever a distance update takes place. Whenever the algorithm updates the distance label of a node from k_1 to k_2, we update *DLIST* (k_1) and *DLIST* (k_2) and check whether *FIRST(k)* = 0. If so, then all nodes in the sets *DLIST* (k_1+1), *DLIST* (k_1+2), have become disconnected from the sink. We scan the sets *DLIST* (k_1+1), *DLIST* (k_1+2), and *mark* all the nodes in these sets so that they are never examined. These nodes are deleted from *DLIST*(·) and *SLIST*(·), and are never stored again in these sets. We then continue with the algorithm until there are no active nodes that are unmarked. At this time we initiate the second phase.

In the second phase of the algorithm, we return the excess of all nodes back to the source. We perform the breadth-first-search from the source to compute the initial distance labels (in the second phase, a distance label d(i) represents a lower bound on the length of the shortest path from node i to node s in the residual network). We then perform preflow-push steps on active nodes until there are no more active nodes. It can be shown that the active nodes can be examined in any order and still the second phase would terminate in O(nm) time. We experimented with several rules for examining active nodes and found that the rule that always examines an active node with highest distance label leads to minimum number of pushes in practice. We incorporated this rule in the algorithm.

4.3(b) FIFO preflow-push algorithm

The FIFO preflow-push algorithm examines active nodes in the first-in-first-out order. The algorithm maintains the set LIST as a queue. It selects a node i from the front of the LIST, performs pushes from this node, and adds newly active nodes to the rear of the LIST. The algorithm examines a node i until either it becomes inactive or it is relabeled. In the latter case, we add node i to the rear of the queue. The algorithm terminates when the queue of active nodes becomes empty.

To analyze the worst-case complexity of the FIFO algorithm, we partition the total number of node examinations into different phases. The first phase consists of the nodes that become active during the *pre-process* step. The second phase consists of all nodes in the queue after all the nodes in the first phase have been examined. Similarly, the third phase consists of all the nodes in the queue after all the nodes in the second phase have been examined, and so on. It can be shown that there are at most $2n^2+n$ phases in the algorithm. Each phase examines any node at most once and each node examination performs at most one non-saturating push. Hence the FIFO preflow-push algorithm performs $O(n^3)$ non-saturating pushes and runs in the same time.

We implemented the FIFO preflow-push algorithm in the same manner as we did the highest label preflow-push algorithm. We implemented the two phase version of the algorithm. However, instead of maintaining active nodes in the lists *SLIST*(·), we maintained them in a circular queue which was stored as an array.

4.3(c) Wave Algorithm

The wave algorithm is a hybrid version of the highest label and FIFO preflow-push algorithms and is due to Ahuja and Orlin [unpublished]. The wave algorithm performs passes over active nodes. Let *NOW* denote the set of active nodes at the beginning of a pass and *NEXT* be an empty set. During the pass, the algorithm examines all nodes in *NOW* in the non-increasing order of distance labels. During a node examination, flow is pushed from the node until either its excess becomes zero or it is relabeled. All relabeled nodes in this pass are added to *NEXT*. At the beginning of the next pass we set $NOW = NEXT$, $NEXT = \phi$ and repeat this process. The algorithm terminates when during a pass no node is relabeled. Observe that if during a pass no node is relabeled then all the excess reaches the sink or the source. Hence there would be O(n^2) passes and O(n^3) node examinations.

We implemented the wave algorithm in the same manner as we implemented the highest label preflow-push algorithm. We tested two phase version of the algorithm. To store active nodes, we maintain the lists *SLIST(k)* consisting of all active nodes with distance label equal to k. We also maintain the variable *level* which represents an upper bound on the highest distance label of an active node in *NOW*. To determine a node with highest distance label in NOW, we sequentially examine the lists *SLIST(level), SLIST(level)–1, ...,* until we find a non-empty list. However, whenever the distance label of a node increases, we update the lists *SLIST*(·) but do not update *level*. Consequently, the sets *SLIST(level), SLIST(level–1), ..., SLIST(1)* implicitly store the set *NOW* and the sets *SLIST(level)+1, ..., SLIST(n)* implicitly store the set *NEXT*. At the beginning of a pass we set *level* = n which essentially amounts to setting *NOW* = *NEXT* and *NEXT* = ϕ.

4.3(d) Lowest Label Preflow-push Algorithm

The lowest label preflow-push algorithm always pushes flow from a node with smallest distance label. This algorithm performs O(n^2m) non-saturating pushes and runs in the same time. This running time is clearly not as attractive as that of the other preflow-push algorithms we tested. The primary objective for testing this algorithm was that the excess scaling algorithm we discuss in the next section can be regarded its scaling variant and we wanted to observe the reduction in the number of non-saturating pushes due to the scaling technique.

We implemented this algorithm in the same manner as we did the highest label preflow-push algorithm. We tested the two phase version of the algorithm. We stored the active nodes in the lists *SLIST*(·) and maintained the variable *level* which represented a lower bound on the lowest distance label of any active node. In order to determine a node with smallest distance label, we examine the lists *SLIST(level), SLIST(level+1), ...,* until we find a non-empty list.

4.4 COMPUTATIONAL RESULTS FOR KARZANOV'S ALGORITHM

The non-saturating pushes and the arc scans for constructing layered networks are the representative operations for the Karzanov's algorithm. Apart from these, we kept a record of the saturating pushes and the number of layered networks constructed.

We use total number of pushes instead of non-saturating pushes as a representative operation.

The number of pushes performed and the arc scans for constructing layered networks are shown in Figures 4.1, 4.2, and 4.5 respectively. We observe that the arc scans for constructing layered networks have a rising share in the sum of the representative operation counts and hence is the asymptotic bottleneck operation [see Figures 4.4 and 4.6]. In Figure 4.7, we observe that Karzanov's algorithm constructs a large number of layered networks for the random grid network and for the random layered network, the increase in the number of layered networks constructed with the size of the network, is not always proportional.

The non-saturating pushes increase greatly with an increase in the size of both the networks and comprise about 90% of the total pushes for both the type of networks of size >= 7,000 [see Figures 4.8]. This is one reason for the slower running time of Karzanov's algorithm as compared to the preflow-push algorithms. A comparison of different preflow-push algorithms, in Section 4.6, further explains the faster running time of preflow-push algorithms.

An estimate of the virtual running time for the Karzanov's algorithm for an instance I was made using the expression:

$$V(I) = C_P \text{ (total pushes)} + C_L \text{(arc scans for constructing layered networks)},$$

where C_P and C_L are constants. We use multiple regression analysis to estimate C_P and C_L for networks of the same density but different sizes.

Network	Density (d)	C_P (10^{-6})	C_L (10^{-6})
Random Layered Network	4	12	7.6
	6	11	7.6
	8	10	7.6
	10	11	7.3
Random Grid Network		17	6.1

Table 4.1. Regression coefficients in virtual running time estimation for Karzanov's algorithm.

As evident from Figure 4.9, the virtual running time is a good approximation for the CPU time. The error in all of the 45 cases is less than 3% and the error is less than 1% for networks larger than 5000 nodes.

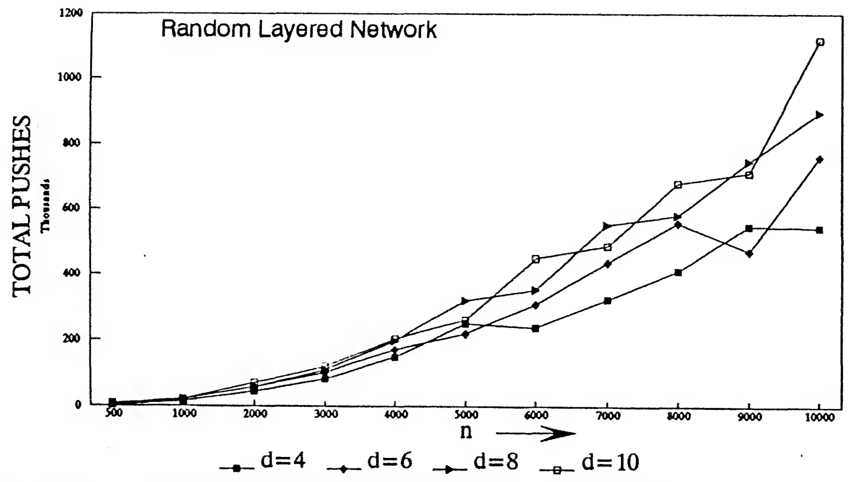


Figure 4.1. Representative Operation: total pushes

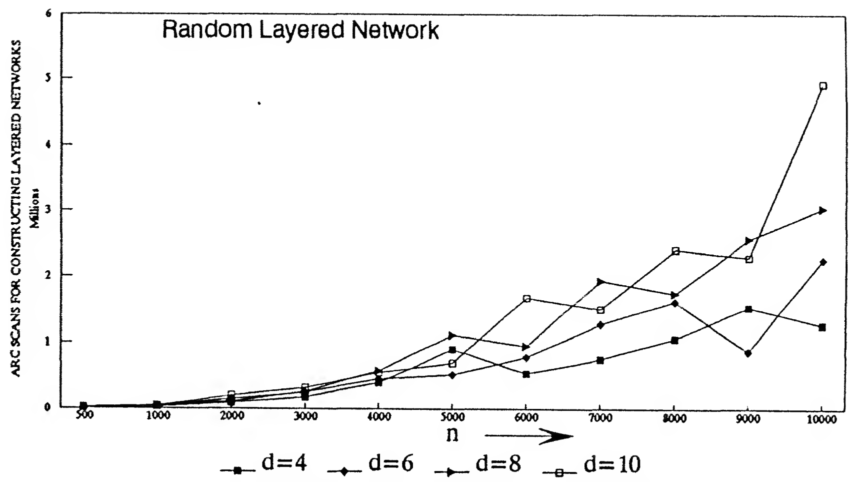


Figure 4.2. Representative Operation: arc scans for constructing layered networks

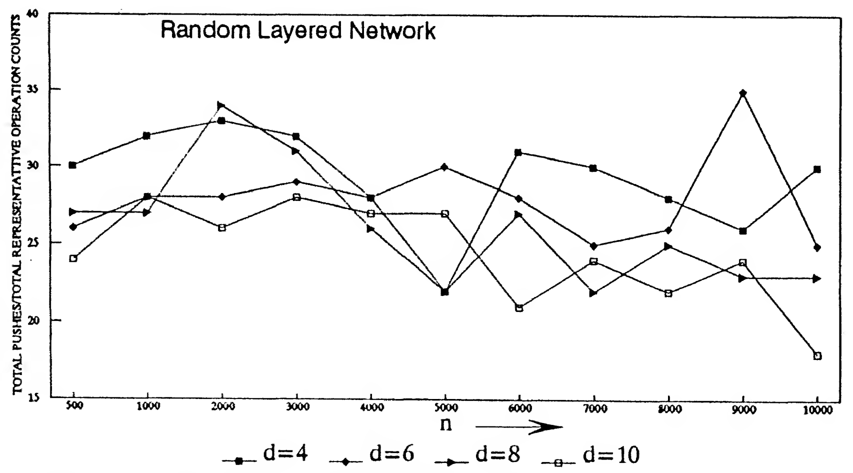


Figure 4.3. Identifying asymptotic bottleneck operation: ratio of total pushes to total representative operation counts

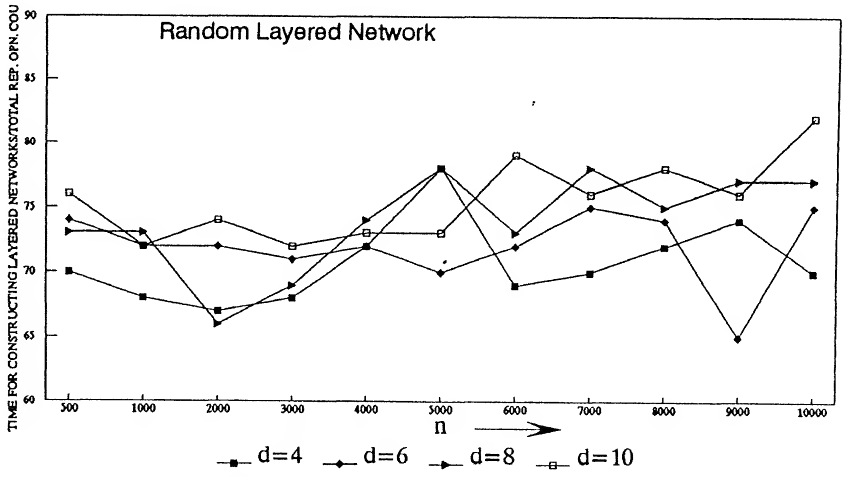


Figure 4.4. Identifying asymptotic botleneck operation: ratio of arc scans for constructing layered networks to total operations

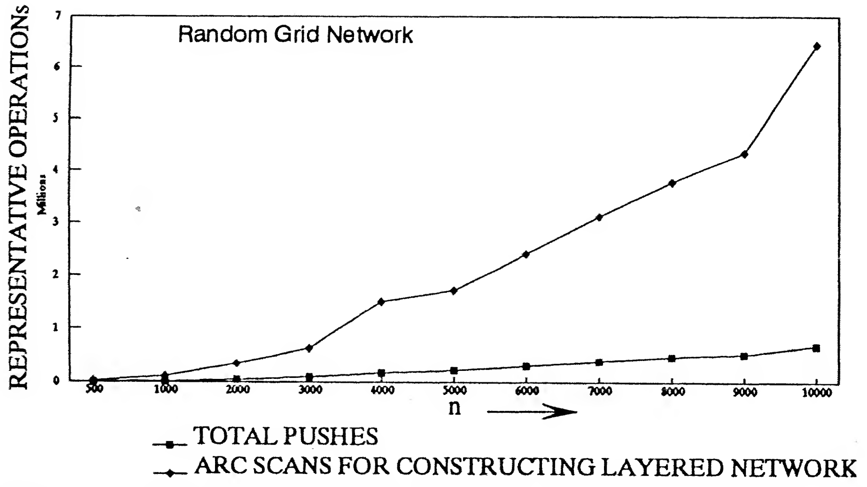


Figure 4.5. Representative operations: total pushes and arc scans for constructing layered networks

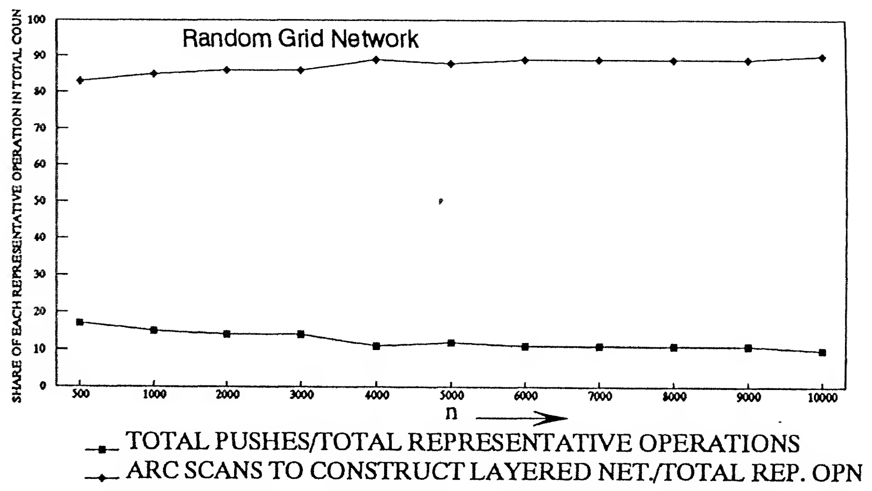


Figure 4.6. Identifying asymptotic botleneck operation: time for pushes and constructing layered networks

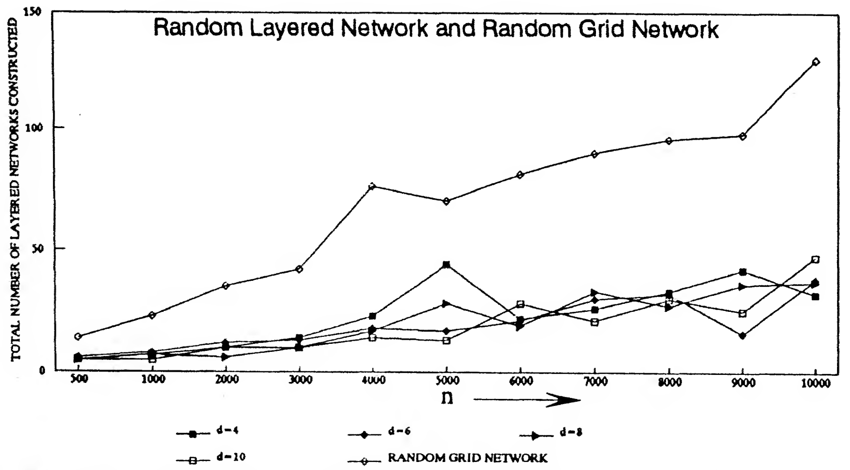


Figure 4.7. Number of layered networks constructed

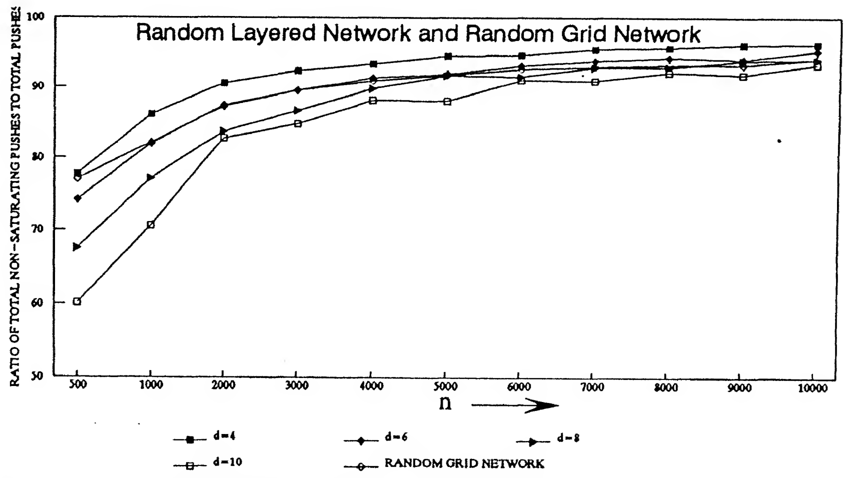


Figure 4.8. Proportion of non-saturating pushes in total pushes

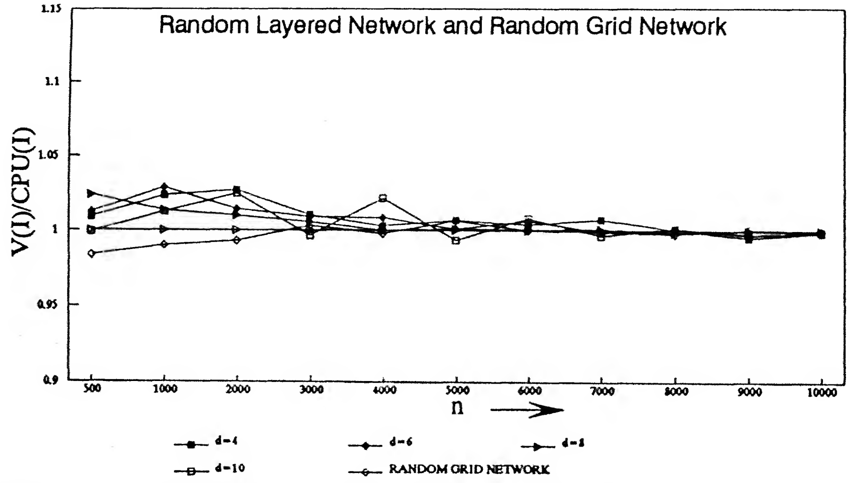


Figure 4.9. Virtual running time estimates CPU running time

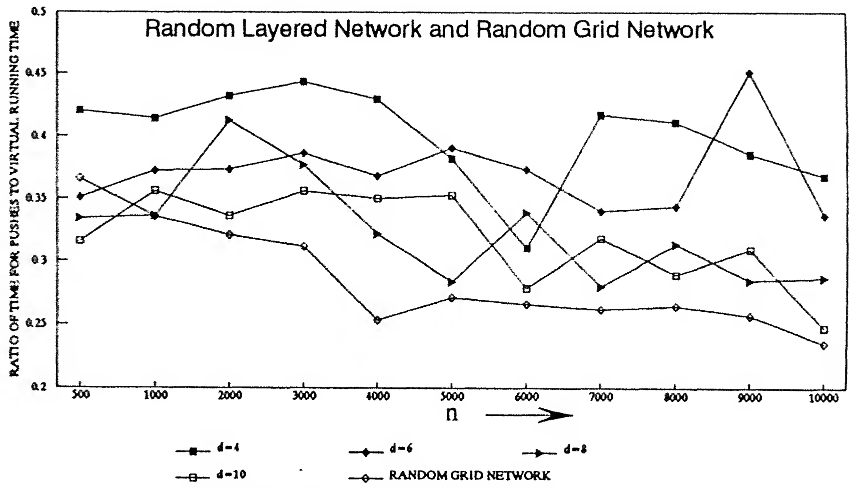


Figure 4.10. Percentage of virtual running time accounted by pushes

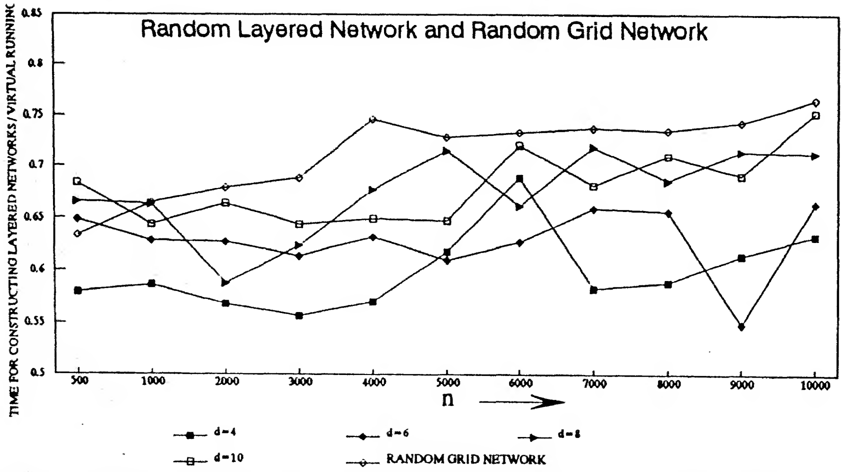


Figure 4.11. Percentage of virtual running time accounted by time to construct layered networks

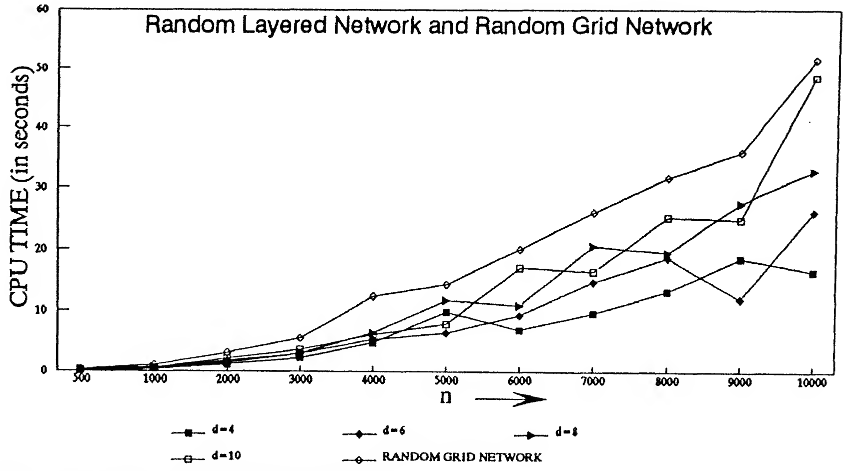


Figure 4.12. CPU time taken

KARZANOV'S ALGORITHM

We found that arc scans for constructing layered networks is the bottleneck operation for small as well as large networks since they account for about 55% to 75% of the virtual running time [see Figure 4.10]. The time for pushes takes about 25% to 45% of the virtual running time [see Figure 4.11] and their decreasing slope establishes that the time to construct layered networks is the asymptotic bottleneck operation for Karzanov's algorithm.

Figure 4.12 shows the CPU time taken by Karzanov's algorithm. We can observe by comparing this with Figure 3.33 that Karzanov's algorithm is about twice faster than Dinic's algorithm for the random layered network and has a comparable time for the random grid network. Comparing Figures 4.7 and 3.28, we observe that Dinic's algorithm constructs about 20% to 30% lesser number of layered networks than Karzanov's algorithm.

We tried to estimate the growth rate of the representative operations of this algorithm as a function of n^{γ}. We plotted log(operation count)/log(n) for different networks. We observe in Figure 4.13 that the total pushes vary between $n^{1.35}$ and $n^{1.5}$. We can see that the non-saturating pushes vary between $n^{1.3}$ and $n^{1.5}$ [see Figure 4.15] and the saturating pushes vary between $n^{1.1}$ and $n^{1.3}$ [see Figure 4.17]. The time to construct layered networks varies between $n^{1.5}$ and $n^{1.7}$ and the number of layered networks constructed varies between $n^{0.25}$ and $n^{0.35}$ for the random layered network and is around $n^{0.5}$ for the random grid network [refer Figures 4.14 and 4.16].

4.5 COMPUTATIONAL RESULTS FOR PREFLOW-PUSH ALGORITHM

The non-saturating pushes and arc scans for updating distance labels of nodes are the representative operations for the preflow-push algorithms. The arc scans for relabeling nodes dominates the number of operations the algorithm performs in updating the current arc and in making saturating pushes. We noted the number of saturating and non-saturating pushes in the first phase to construct a maximum preflow and for the second phase in which the maximum preflow is converted into a maximum flow. We also counted the total number of relabels and the arc scans for relabels in both the phases.

(a) FIFO VERSION

For the first-in-first-out version of the preflow-push algorithm, the total pushes and the arc scans for relabels, in general, increase with the size and density of the random layered network [see Figures 4.18 and 4.19]. As in the algorithms discussed in previous sections, the relabels and in some cases the pushes too vary erratically .

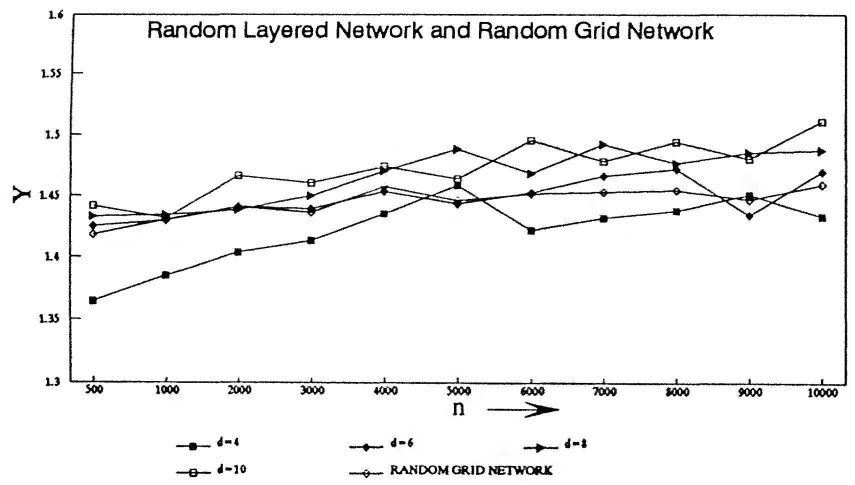


Figure 4.13. Growth rate of total pushes
Y = log(total pushes) / log(n)

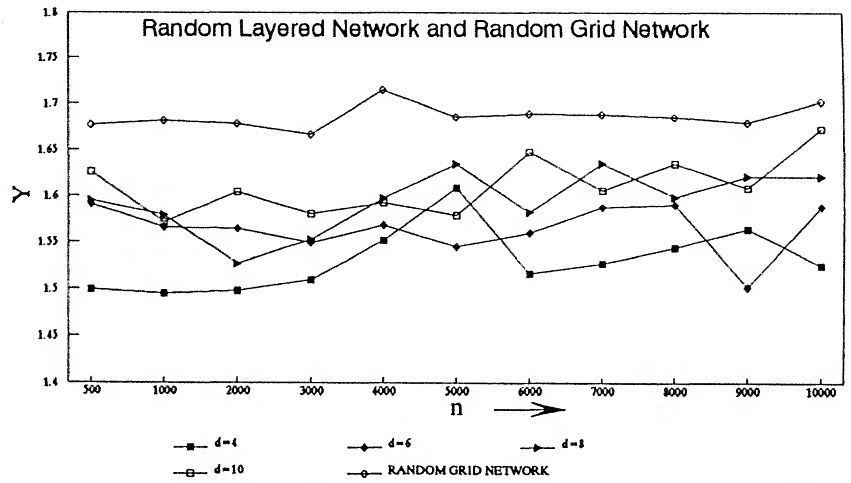


Figure 4.14. Growth rate of arc scans for layered networks
Y = log(arc scans for layered networks) / log(n)

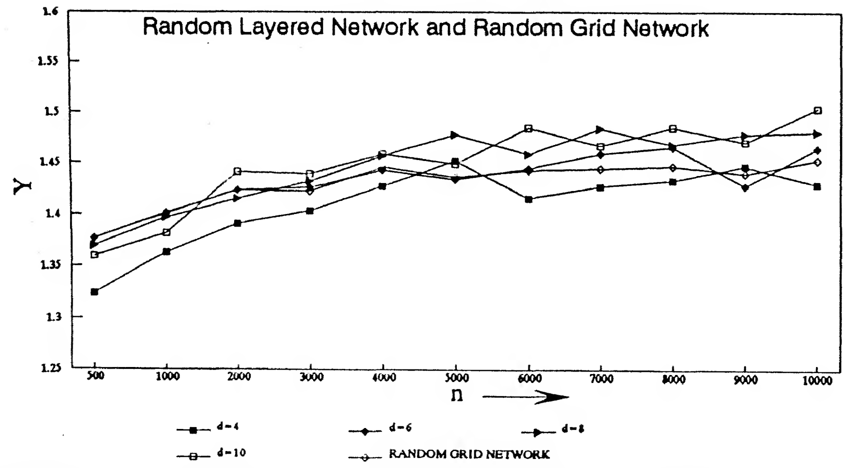


Figure 4.15. Growth rate of non–saturating pushes
Y = log(# of non–saturating pushes) / log(n)

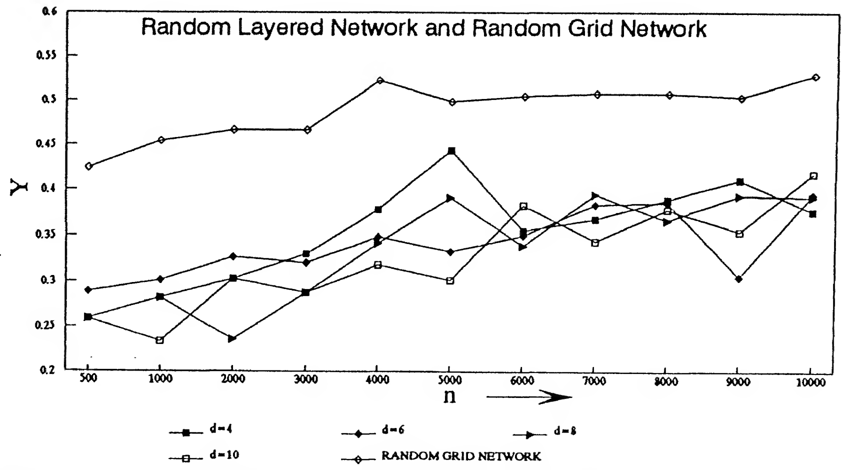


Figure 4.16. Growth rate of layered networks
Y = log(# of layered networks) / log(n)

KARZANOV'S ALGORITHM

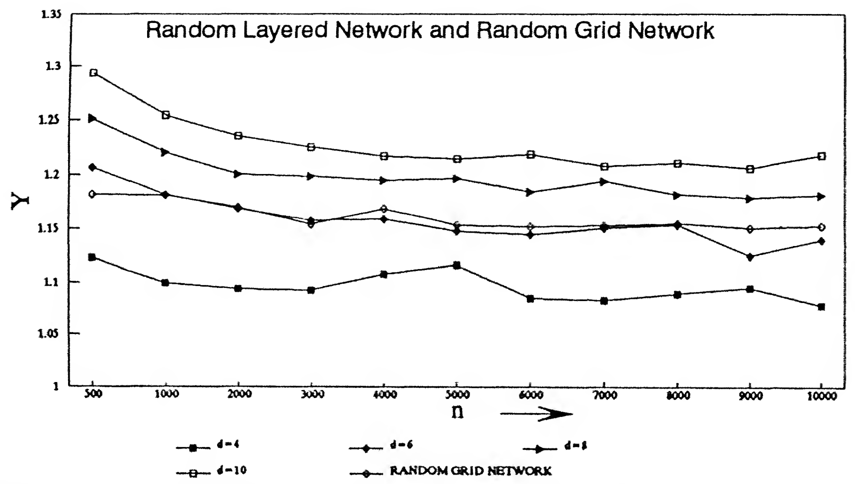


Figure 4.17. Growth rate of saturating pushes
Y = log(# of saturating pushes) / log(n)

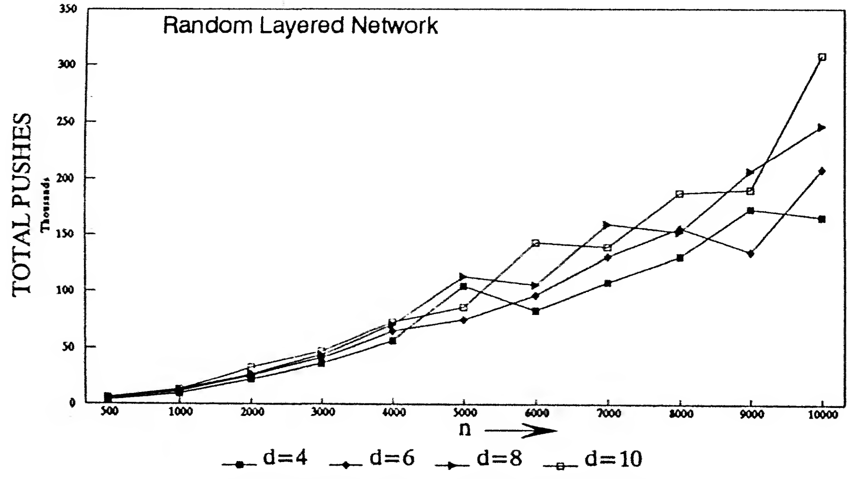


Figure 4.18. Representative operation: total pushes

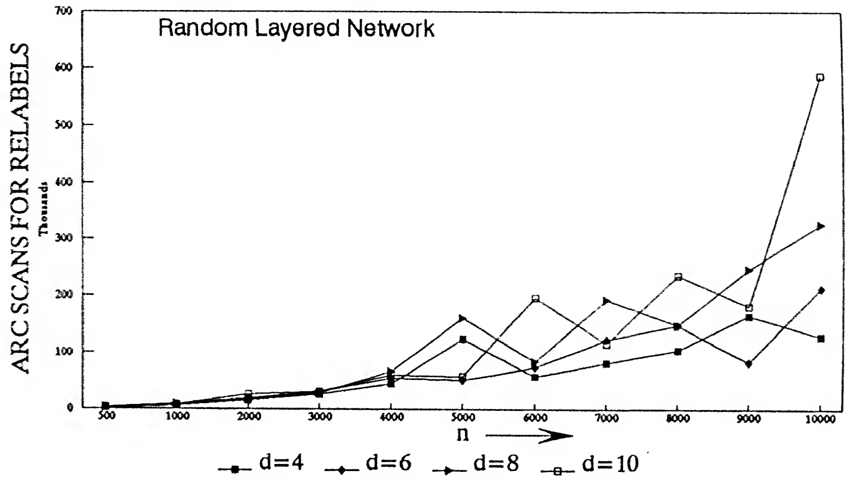


Figure 4.19. Representative Operation: Arc scans for relabels

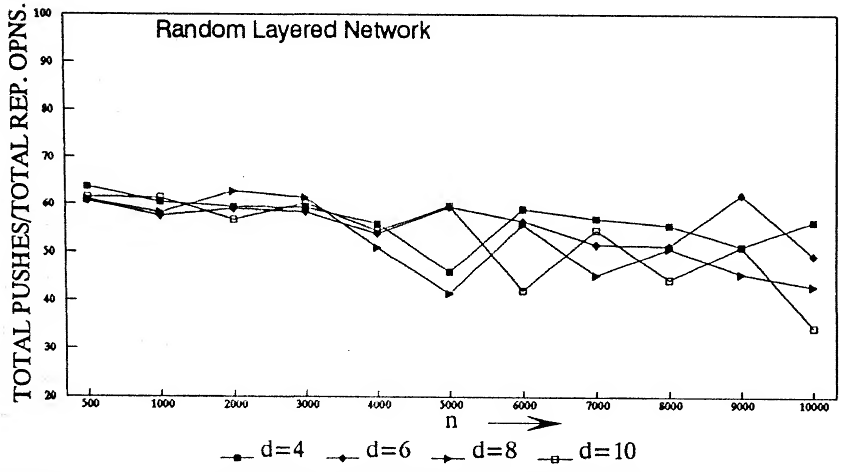


Figure 4.20. Identifying asymptotic bottleneck operation: ratio of total pushes to total representative operation counts

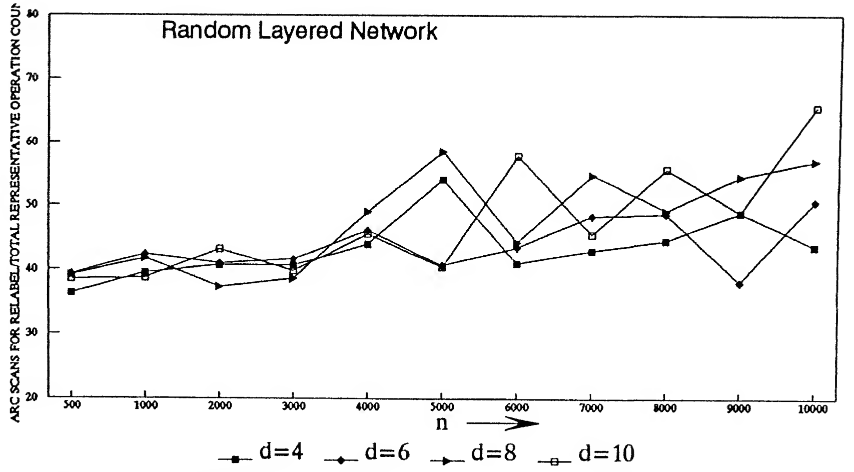


Figure 4.21. Identifying asymptotic bottleneck operation: ratio of arc scans for relabels to total representative opn. count

PREFLOW–PUSH ALGORITHM – FIFO VERSION

They increase fairly smoothly with the size of the random grid network [see Figure 4.22]. The arc scans for relabels have an increasing share in the sum of the representative operations counts for this algorithm [see Figures 4.20, 4.21, 4.23]. Hence it is the asymptotic bottleneck operation for the FIFO version of the preflow-push algorithm.

We observe in Figure 4.24 that 70%-85% of the pushes on the random grid network are non-saturating. For the random layered network, about 50% to 80% of the pushes are non-saturating and the proportion of non-saturating pushes in the total pushes decreases with density. Thus, for a given size, sparse networks have more non-saturating pushes then dense networks.

We state in Section 4.3(b) that we divide the total number of node examinations into phases. Figure 4.25 shows the number of phases for each problem size for a random layered network of density 6. It is a representative curve. While in the worst-case there can be $2n^2+n$ phases in the algorithm, we find that in practice they are far less. For example, in this case there are less than 0.1n phases.

An estimation of the CPU time using virtual running time was made using the expression:

$$V(I) = C_P \text{ (total pushes)} + C_R \text{(arc scans for relabels)},$$

where C_P and C_R are constants. We use multiple regression analysis to estimate C_P and C_R for networks of the same density but diferent sizes.

Network	Density (d)	C_P (10^{-6})	C_R (10^{-6})
Random Layered Network	4	17	7.1
	6	17.3	7.1
	8	17	7.8
	10	16.7	8.3
Random Grid Network		17.3	7.9

Table 4.2. Regression coefficients in virtual running time estimation for FIFO version of the preflow-push algorithm.

Again, it can be seen from Figure 4.26 that the virtual running time estimates the CPU time fairly well. The error is within 4% for all but 2 cases and less than 1% for large networks of $n \geq 5{,}000$. Figures 4.27 and 4.28 show that pushes are the bottleneck operation for random layered network of all the sizes we tested i.e., for $n \leq$

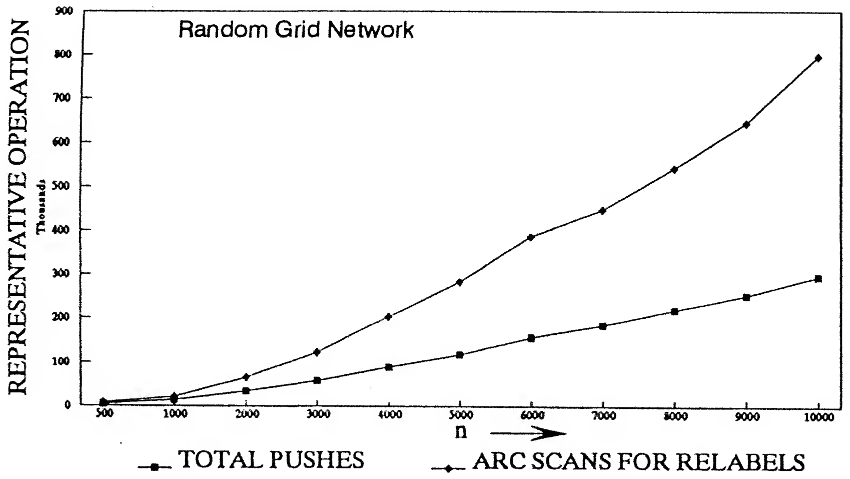


Figure 4.22. Representative operations on random grid network

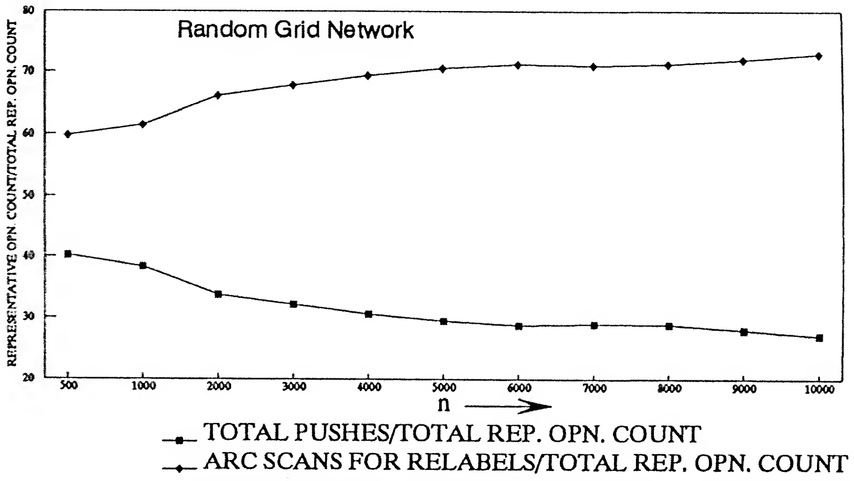


Figure 4.23. Identifying asymptotic bottleneck operation: ratio of representative operation count to total rep. opn. count

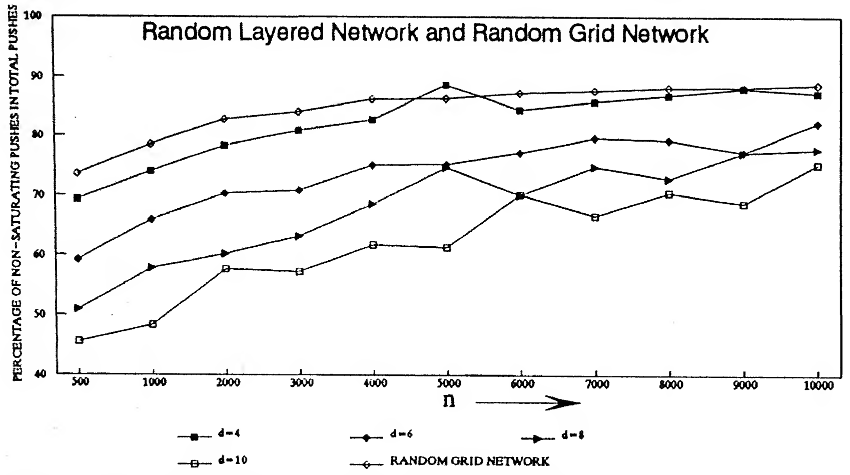


Figure 4.24. Proportion of non-saturating pushes in total pushes

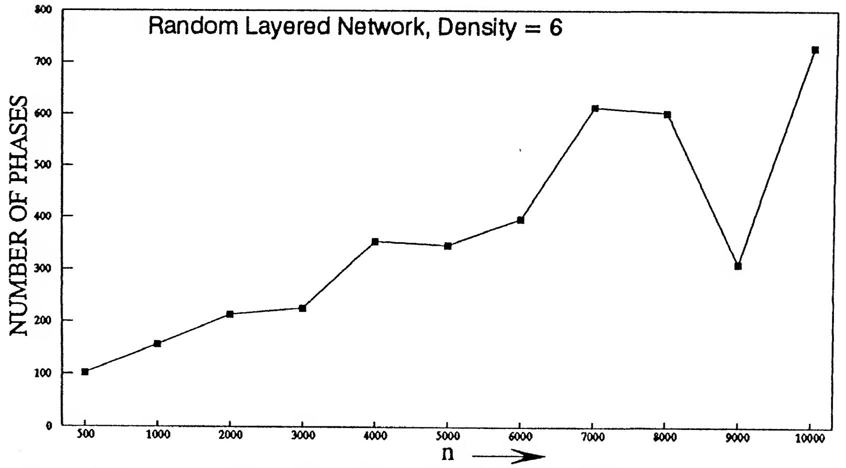


Figure 4.25. Number of phases in the algorithm

PREFLOW–PUSH ALGORITHM – FIFO VERSION

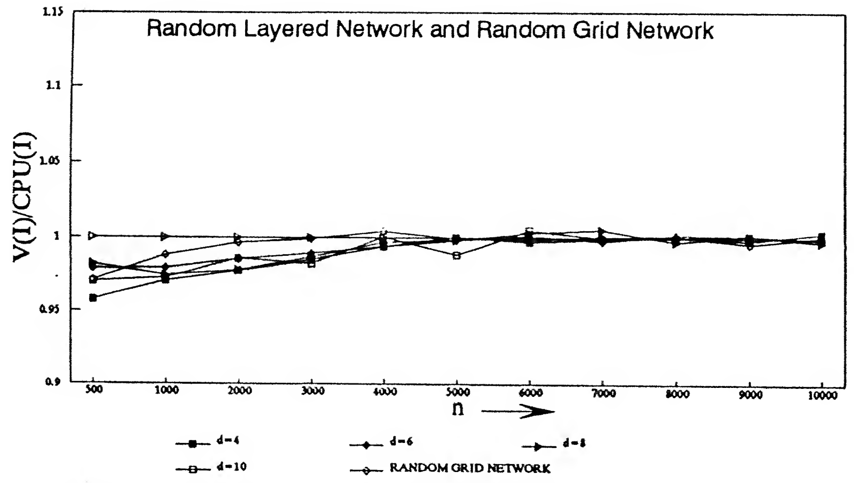


Figure 4.26. Approximation of CPU time by virtual running time

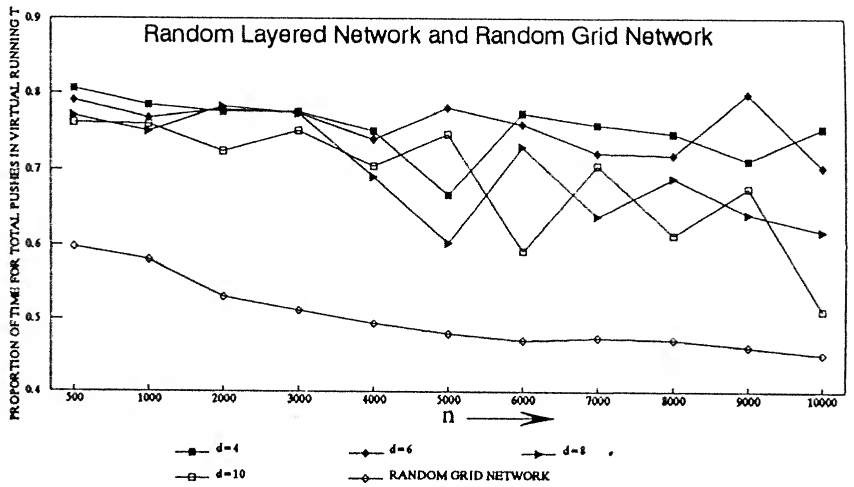


Figure 4.27. Proportion of time for pushes in virtual running time

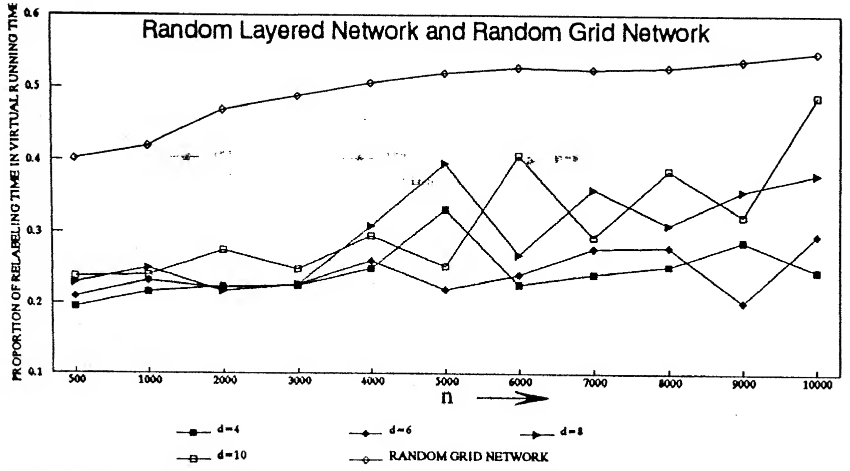


Figure 4.28. Proportion of relabeling time in virtual running time

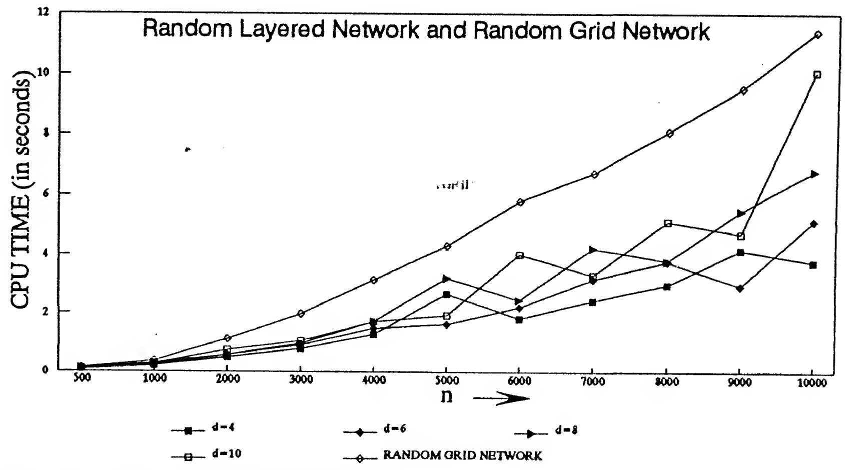


Figure 4.29. CPU time taken

PREFLOW–PUSH ALGORITHM – FIFO VERSION

10,000, but for the random grid network they account for a greater share of the virtual running time for only networks of size less than 4,000 nodes, beyond which the arc scans for relabels are the bottleneck operation. We can infer from the slope of the curves in these figures and in Figures 4.20, 4.21, and 4.23 that, in the asymptotic case, the relabeling time is the bottleneck operation. Figure 4.29 shows the CPU time taken by the FIFO version of the preflow-push algorithm for different test problems.

We studied the flow sent into the sink as a function of the pushes and node relabels. Studies show that for the FIFO version of the preflow-push algorithm, large portion of the total possible flow into the sink is sent using a fairly small proportion of the total node relabels. While no flow reaches the sink for a considerable number of relabels (17% to 60 %, smaller percentage for larger network sizes), a large percentage of the total possible flow (20%-50%, lower percentage for smaller networks) reaches the sink using a small percentage of the relabels. The remaining flow reaches the sink after performing a large number of node relabels. Figure 4.30 shows the number of relabels performed as flow is sent to the sink in a large size network. The number of pushes used to send the flow into the sink also vary similarly but a large percentage of total pushes (30% to 70%, lower percentage for larger networks) are necessary to send the initial flow ino the sink [see Figure 4.31].

As described in Section 4.3, we first establish a maximum preflow using the algorithm being tested and then use a version of the highest label preflow-push algorithm to convert this maximum preflow to a maximum flow by pushing the excesses from nodes other than the sink to the source. We observed that less than 0.5n pushes were required in this phase [see, Figure 4.32]. More pushes were required on the random layered network. From Figure 4.33 we can observe that the pushes in the second phase comprise only 2-5% of the total pushes. The number of node relabels in the second phase were also very small.

We tried to estimate the growth rate of the representative operations as a function of n^{γ}. We plot log(operation count)/log(n) for several operations. Figure 4.34 shows that the total pushes for the FIFO version of the preflow-push algorithm varies between $n^{1.3}$ and $n^{1.4}$. The non-saturating pushes vary between $n^{1.26}$ and $n^{1.32}$ for the random layered network and between $n^{1.32}$ and $n^{1.35}$ for the random grid network [see Figure 4.35]. We know that the non-saturating pushes for this version of preflow-push algorithm are $O(n^3)$. The saturating pushes vary between $n^{1.1}$ and $n^{1.3}$ [see Figure 4.36] while there can be at most nm saturating pushes. The number of relabels can at most be $2n^2$. They vary between n and $n^{1.15}$ for the random layered network and between $n^{1.15}$ and $n^{1.25}$ for the random grid network as seen in Figure 4.37. The number of

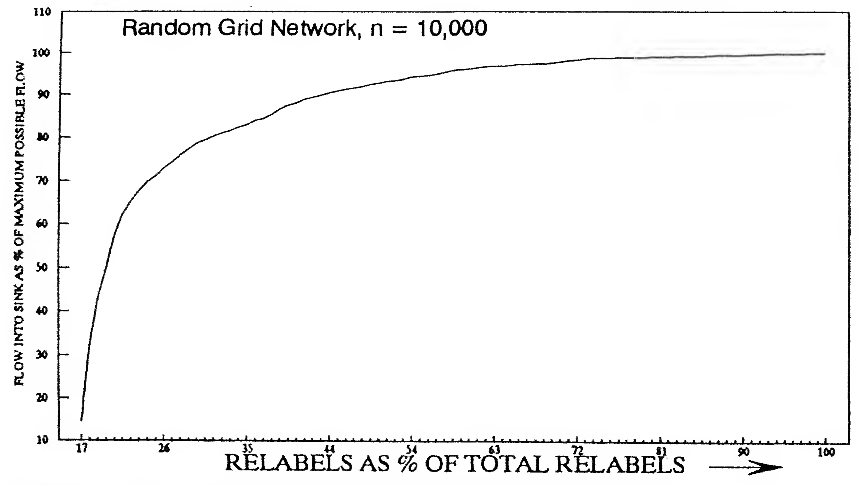


Figure 4.30. Flow into sink vs. node relabels

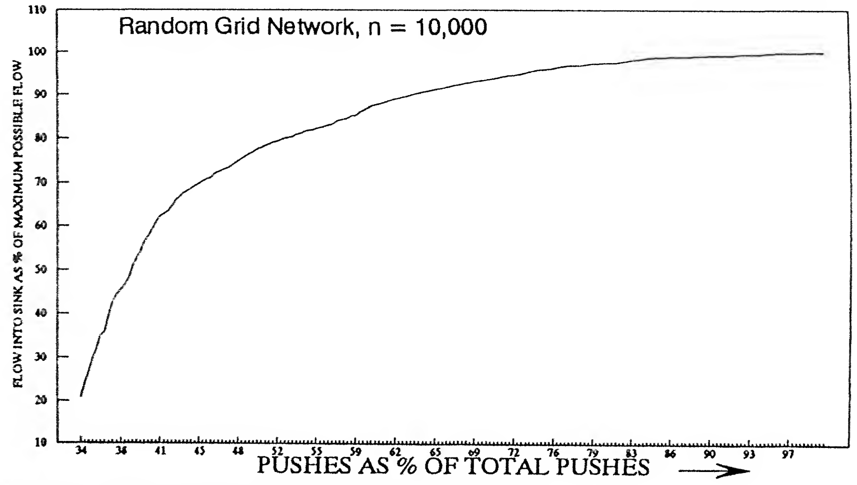


Figure 4.31. Flow into sink vs. pushes

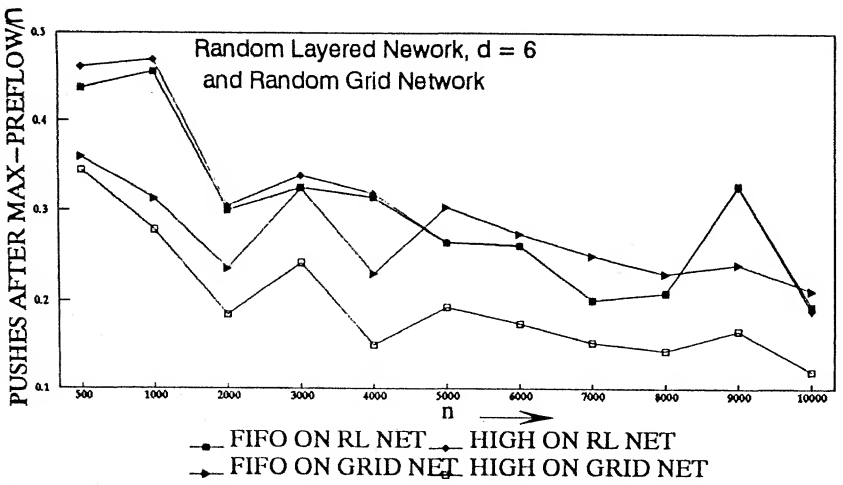


Figure 4.32. Ratio of pushes required to convert maximum preflow into maximum flow to number of nodes in network.

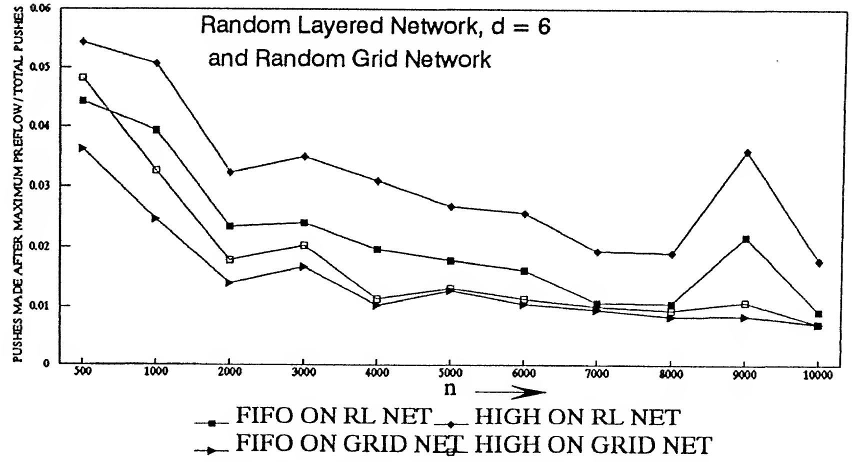


Figure 4.33. Ratio of pushes required to convert maximum preflow into maximum flow to total pushes made.

PREFLOW−PUSH ALGORITHM − FIFO VERSION

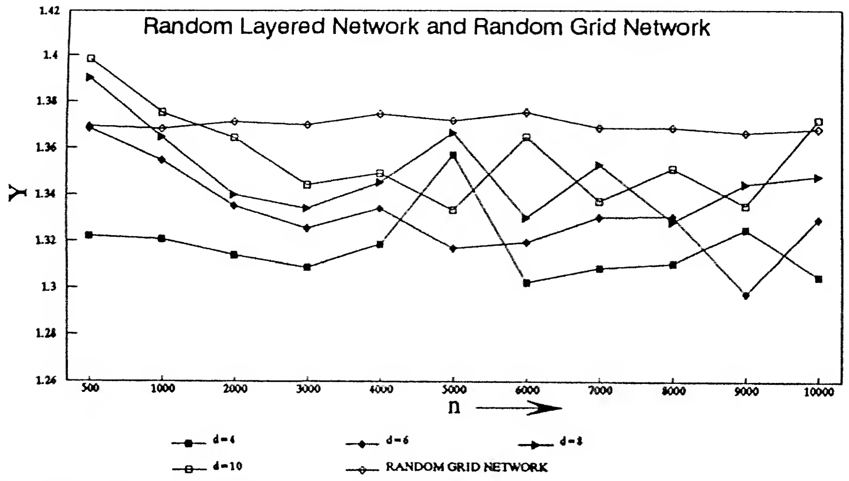


Figure 4.34. Growth rate of pushes
Y = log(total pushes) / log(n)

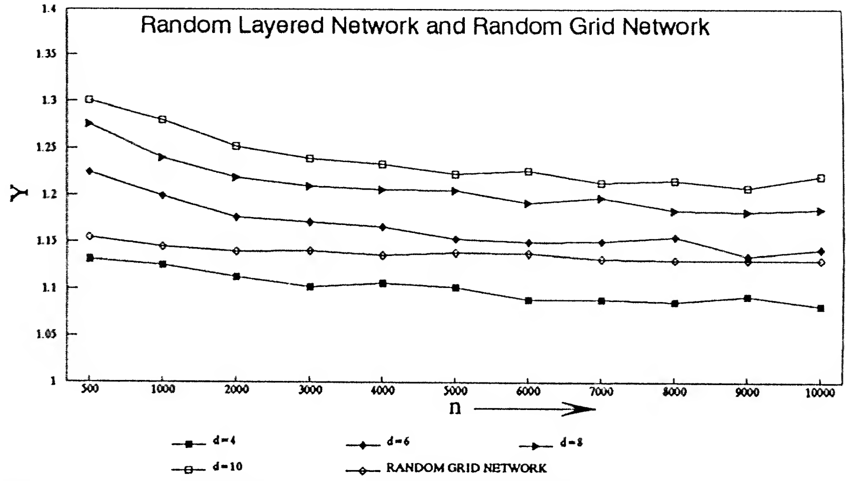


Figure 4.35. Growth rate of saturating pushes
Y = log(saturating pushes) / log(n)

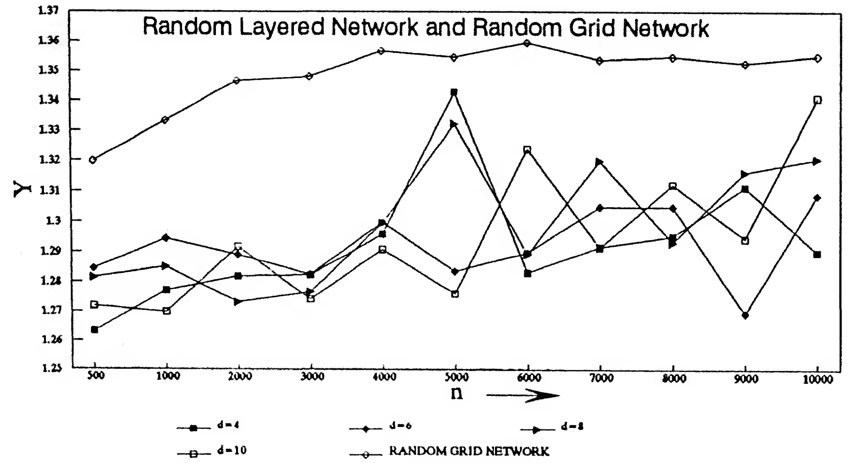
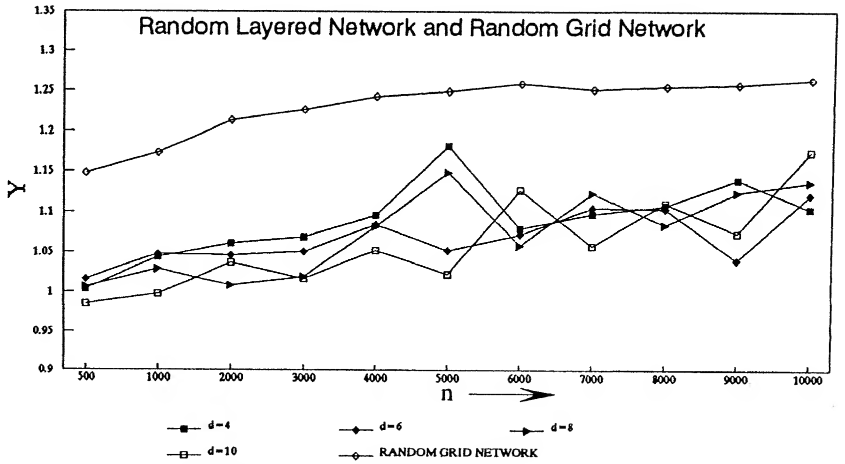


Figure 4.36. Growth rate of non–saturating pushes
Y = log(non–saturating pushes)/ log(n)



Fogure 4.37. Growth rate of relabels
Y = log(# of relabels)/ log(n)

PREFLOW–PUSH ALGORITHM – FIFO VERSION

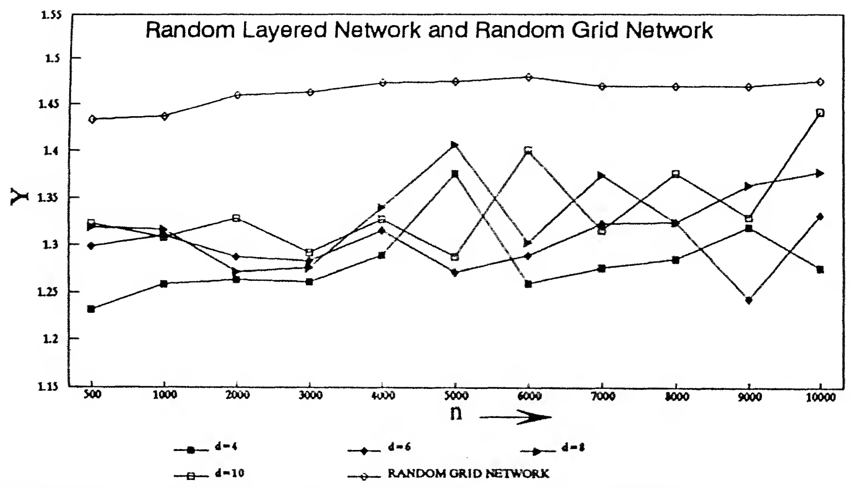


Figure 4.38. Growth rate of arc scans for relabels
Y = log(arc scans for relabels)/ log(n)

arc scans for relabels are $O(n^2m)$ for this algorithm but they vary between $n^{1.25}$ and $n^{1.4}$ for the random layered network and as $n^{1.5}$ for the random grid network as seen from Figure 4.38.

(b) HIGHEST LABEL VERSION

The pushes and arc scans for node relabels, in general, increase with the size and density of the random layered network [Figures 4.39 and 4.40]. On a few instances, they vary erratically. Their increase is uniform for the random grid network [Figure 4.43]. The increase in the arc scans for relabels with the increase in the size and density of the random layered network is erratic and sometimes they decrease too. Their variation is less pronounced than that for other algorithms.

We find that the arc scans for node relabels is the asymptotic bottleneck operation for the highest label version of the preflow-push algorithm [see Figure 4.42 and 4.44] since it has an increasing share in the sum of representative operations.

As seen in Figure 4.45, for the highest label version of the preflow-push algorithm, the saturating pushes comprise about 40% to 60% of the total pushes for the random layered network and are about 25% to 40% for the random grid network. This ratio of saturating pushes to total pushes is superior to the corresponding values for all other preflow-push algorithms we tested.

We observe, in Figure 4.46, that the highest distance label version of the preflow-push algorithm takes much more time to solve a random grid network of a given size than a random layered network of the same size for densities upto 10n. The arc scans for relabels are much more for a random grid network, e.g., there are about 120,000 arc scans for relabels for a random layered network of n=10,000 and d=10. There are about 650,000 relabels for a random grid network of n=10,000. Comparing Figure 3.14 and Figure 4.46, we can see that the random grid network is not harder than the random layered network for the shortest augmenting path algorithm unlike that for the highest distance version of the preflow-push algorithm.

We obtained the virtual running time for this algorithm as a function of the total pushes and the arc scans for relabels.

$$V(I) = C_P \text{ (total pushes)} + C_R \text{ (arc scans for relabels)}.$$

where C_P and C_R are constants. We use multiple regression analysis to estimate C_P and C_R for networks of the same density but diferent sizes.

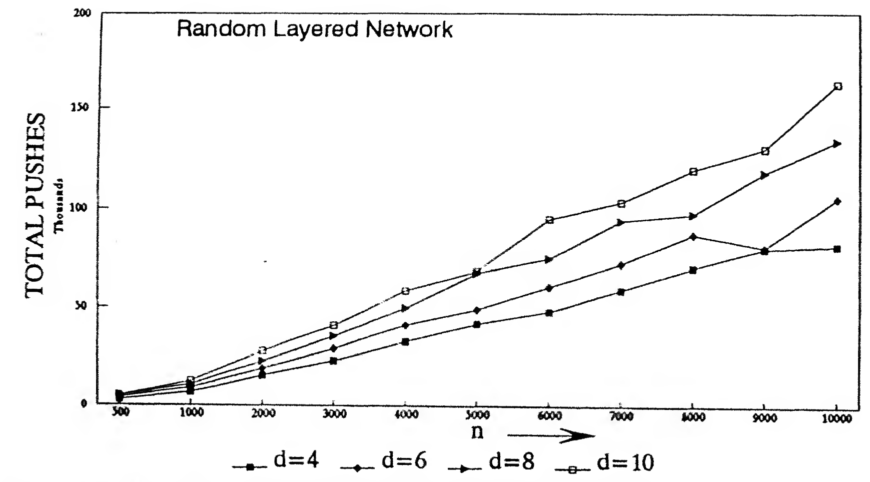


Figure 4.39. Representative Operation: total pushes

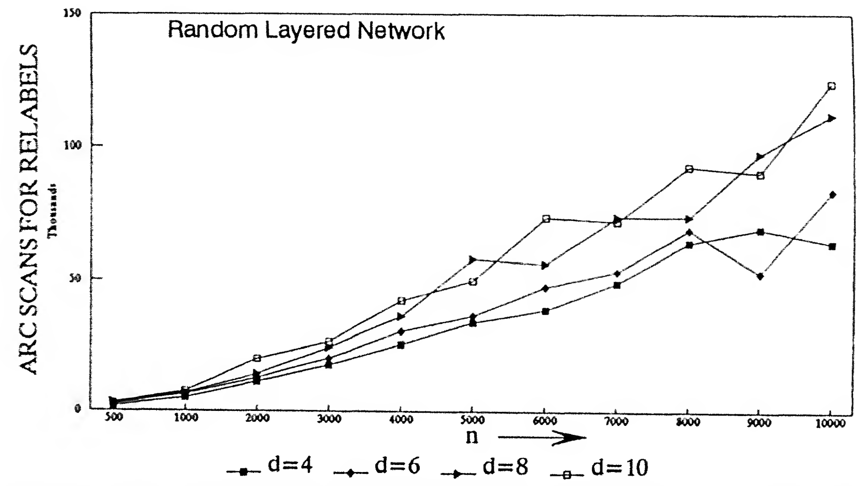


Figure 4.40. Representative operation: arc scans for relabels

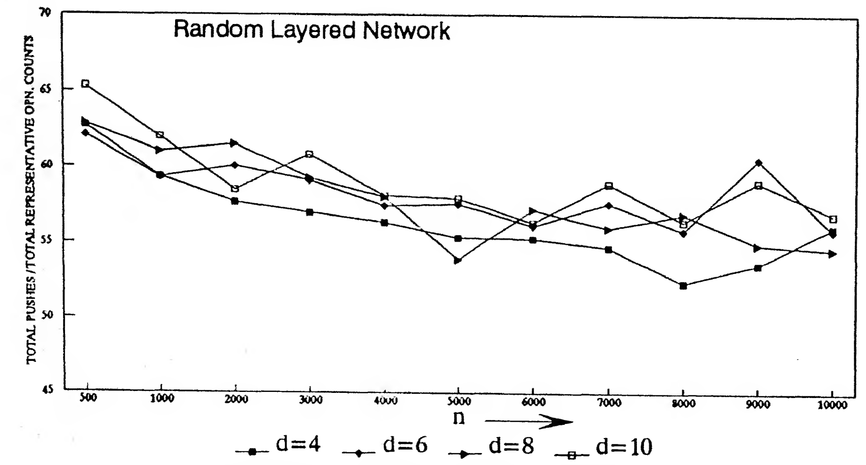


Figure 4.41. Identifying asymptotic bottleneck operation: ratio of pushes to total representative operation counts

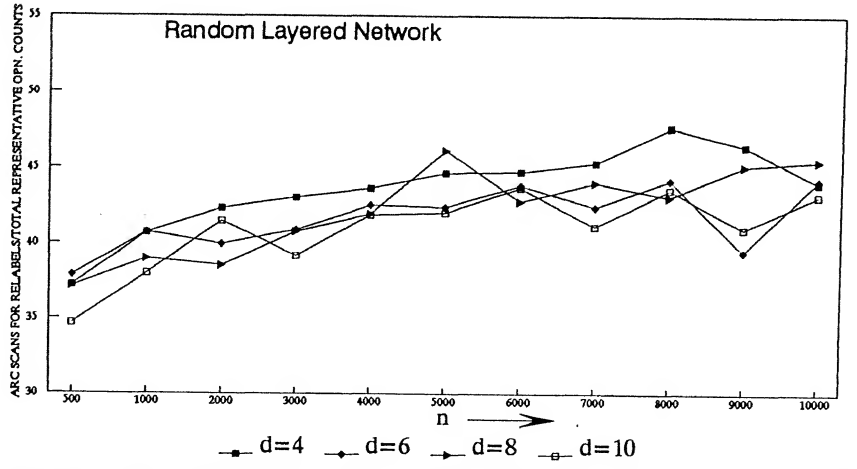


Figure 4.42. Identifying asymptotic bottleneck operation: rati of arc scans for relabels to total representative opn. count

PREFLOW–PUSH ALGORITHM – HIGHEST LABEL VERSION

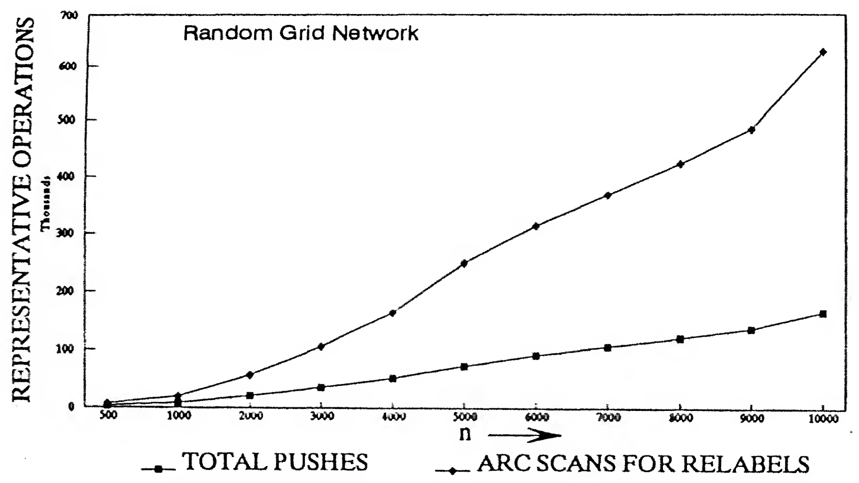


Figure 4.43. Representative operation: total pushes and arc scans for relabeling

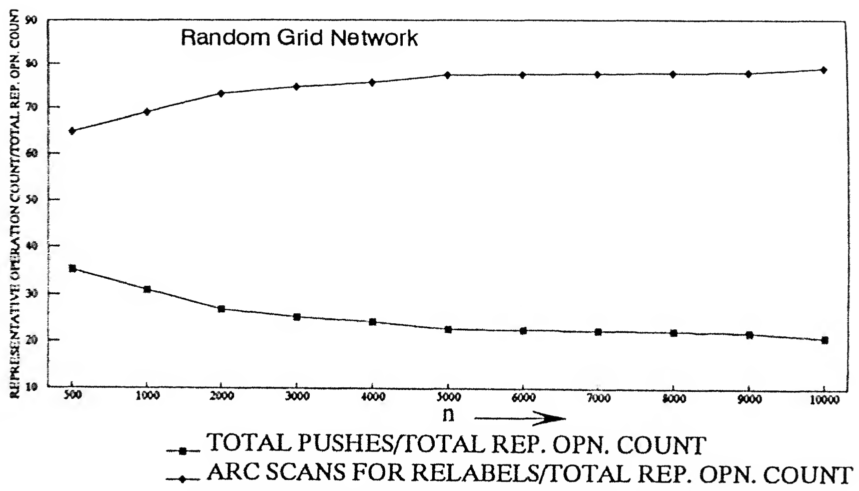


Figure 4.44. Identifying asymptotic bottleneck operation: ratio of representative operation count to total rep. opn. count

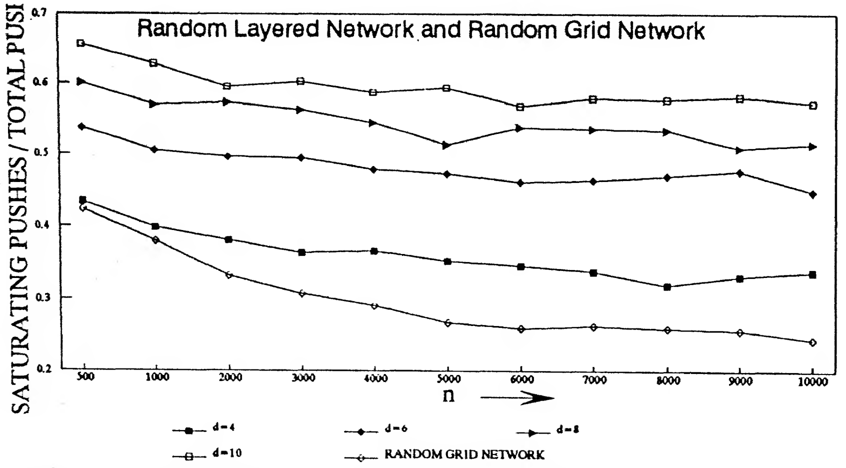


Figure 4.45. Proportion of saturating pushes in total pushes

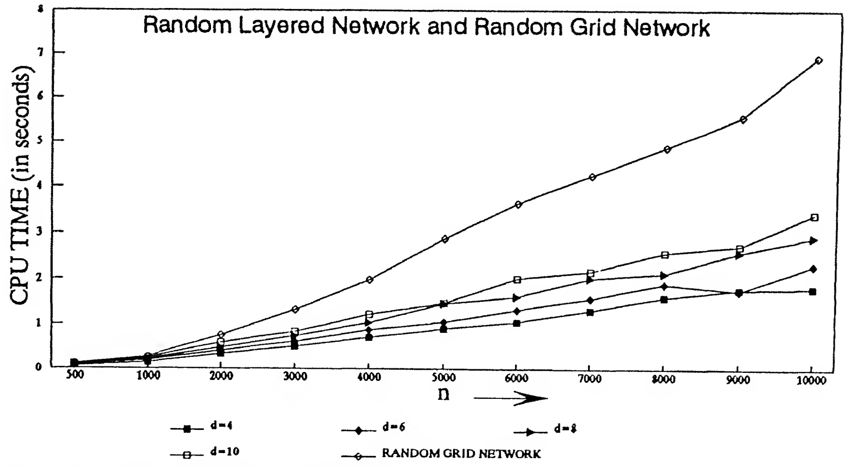


Figure 4.46. CPU time taken

PREFLOW–PUSH ALGORITHM – HIGHEST LABEL VERSION

Network	Density (d)	C_P (10^{-6})	C_R (10^{-6})
Random Layered Network	4	17	5.7
	6	19.9	2.03
	8	18.6	3.48
	10	18.3	3.53
Random Grid Network		19.8	5.7

Table 4.3. Regression coefficients in virtual running time estimation for Highest label version of the preflow-push algorithm.

A plot of V(I)/CPU(I) for different problem instances shows that V(I) closely approximates CPU(I) [see Figure 4.47]. It can be observed that all the cases are within 4% error and more than 50 of the 55 data points are within 2% error. The approximation to the CPU time increases with the size of the network and for $n \geq 6{,}000$, the virtual running time is within 1% of the CPU time. From, Figures 4.48 and 4.49 we can confirm that the time for pushes is the bottleneck operation for the random layered network and random grid network with $n \leq 10{,}000$ and $d \leq 10$, but the relabeling time is the asymptotic bottleneck operation since it has an increasing trend. With an increase in the density of the random layered network, in general, the increase in pushes is greater than the increase in the arc scans for relabels.

In the worst-case, the saturating pushes are bounded by nm and the non-saturating pushes are $O(n^2m^{1/2})$. In Figure 4.50, we observe that the total pushes vary between $n^{1.25}$ and $n^{1.4}$ for the random layered network and are around $n^{1.35}$ for the random grid network. The non-saturating pushes are around $n^{1.2}$ for the random layered network and are between $n^{1.23}$ and $n^{1.28}$ for the random grid network [see Figure 4.51]. The saturating pushes are between $n^{1.15}$ and $n^{1.35}$ as seen in Figure 4.52. The number of relabel operations, in the worst-case, is $2n^2$ and the arc scans to perform relabels operations is O(nm). We observe that while the number of relabel operations vary between n and $n^{1.1}$ for random layered network and between $n^{1.15}$ and $n^{1.25}$ for the random grid network [see, Figure 4.53], the arc scans for relabels vary between $n^{1.2}$ and $n^{1.3}$ for the random layered network and are around $n^{1.45}$ for the random grid network [see Figure 4.54].

We studied the variation of the highest distance label with the pushes and relabels as the maximum preflow is established. The highest distance label decrease gradually with intermittent jumps upwards. Figure 4.55 and 4.56 show this variation for a representative test problem.

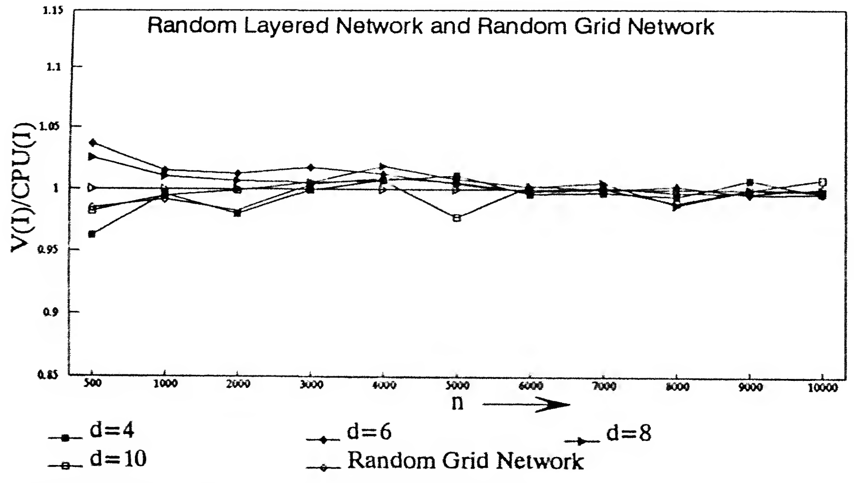


Figure 4.47. Estimation of CPU time by virtual running time

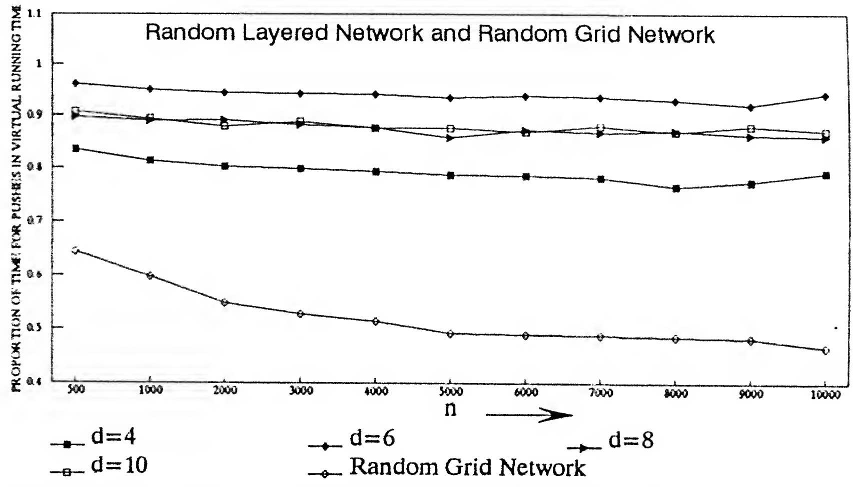


Figure 4.48. Share of time for pushes in virtual running time

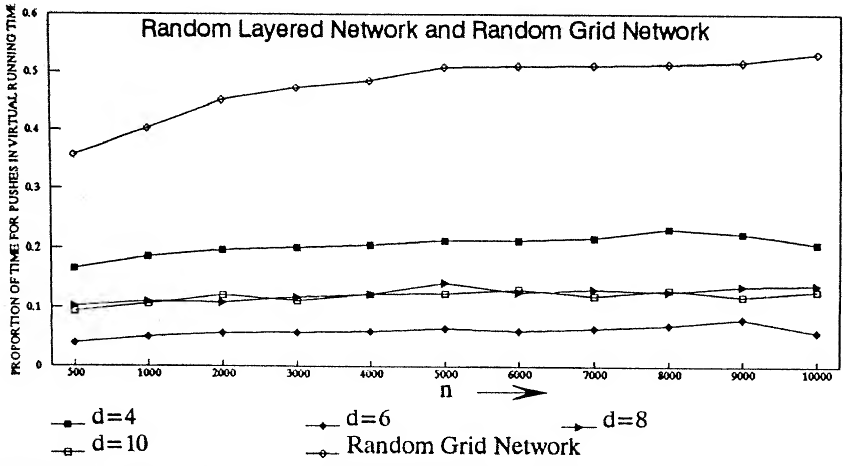


Figure 4.49. Proportion of relabeling time in virtual running time

PREFLOW–PUSH ALGORITHM – HIGHEST LABEL VERSION

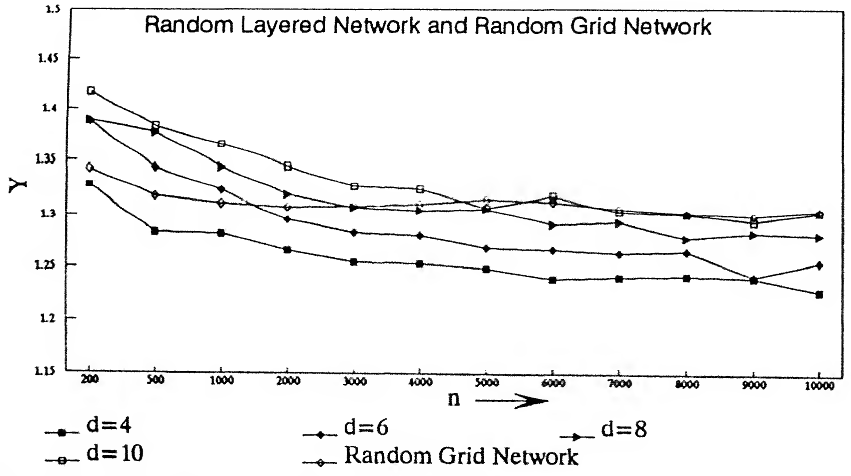


Figure 4.50. Growth rate of total pushes
Y = log(total pushes)/ log(n)

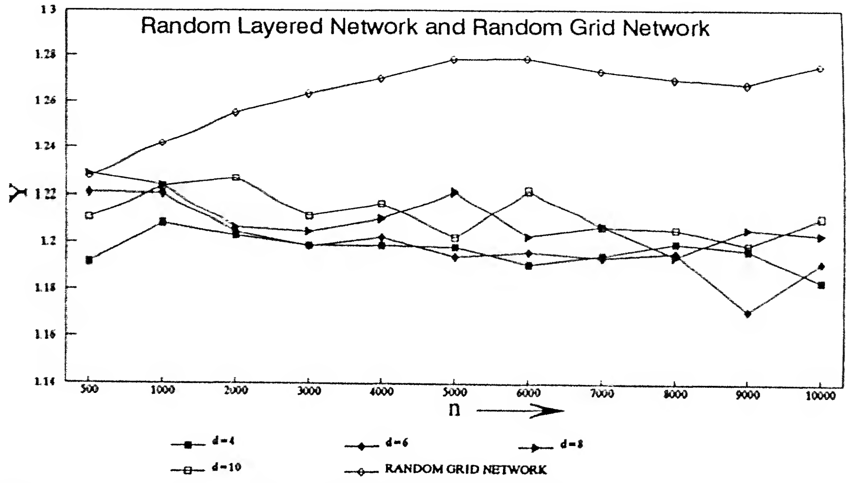


Figure 4.51. Growth rate of non−saturating pushes
Y = log(non−saturating pushes)/log(n)

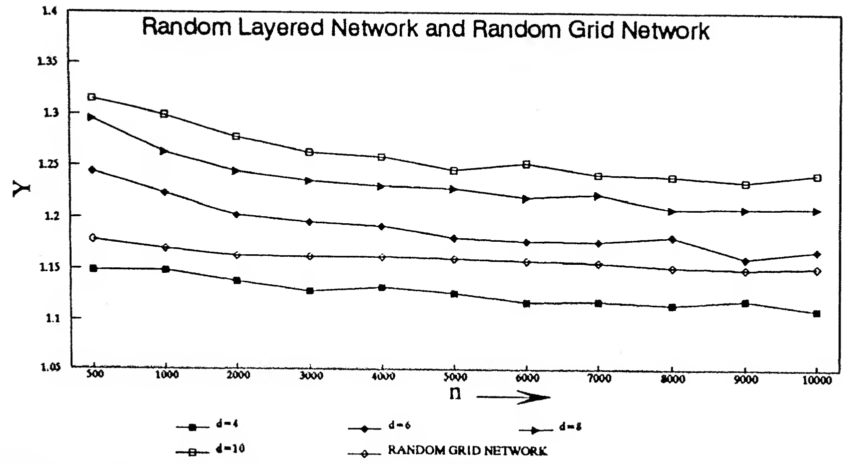


Figure 4.52. Growth rate of saturating pushes
Y = log(saturating pushes)/log(n)

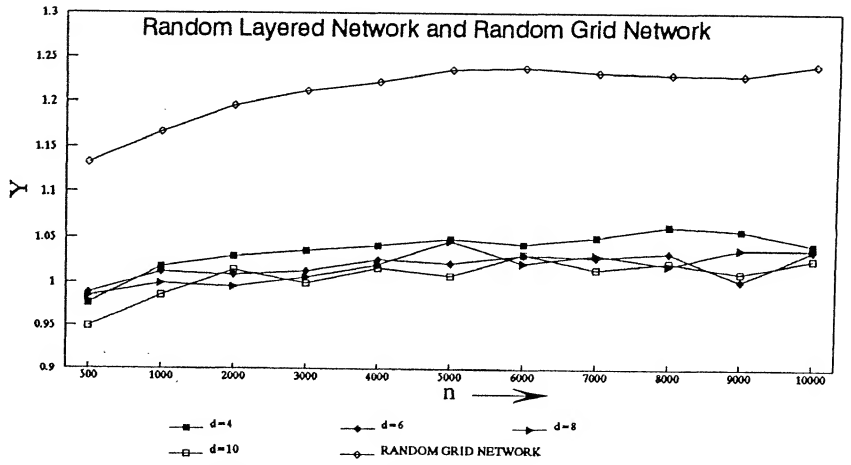


Figure 4.53. Growth rate of relabels
Y = log(# of relabels) /log(n)

PREFLOW−PUSH ALGORITHM − HIGHEST LABEL VERSION

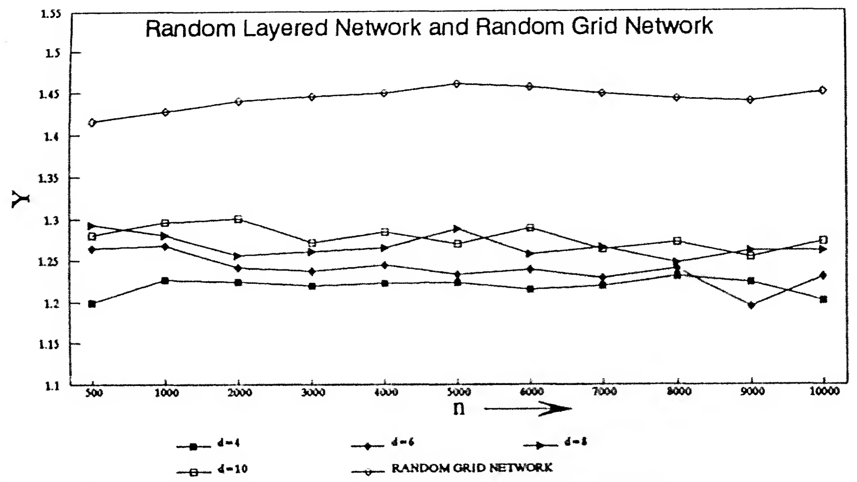


Figure 4.54. Growth rate of arc scans for relabels
Y = log(arc scans for relabels)/log(n)

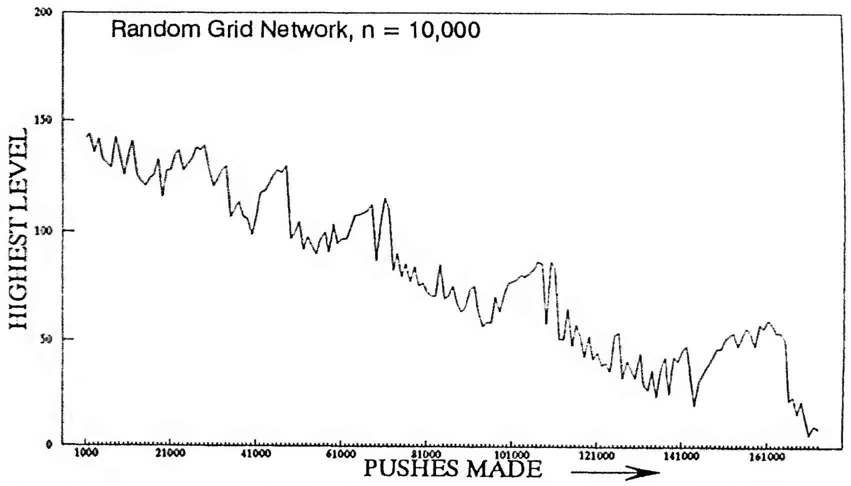


Figure 4.55. Highest distance label in the residual network vs. # of pushes made

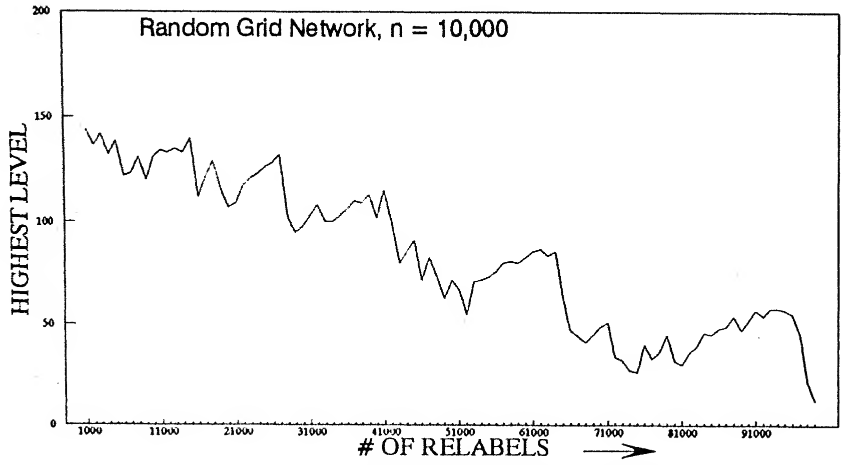


Figure 4.56. Highest distance label in the residual network vs. # of node relabels done

PREFLOW−PUSH ALGORITHM − HGHEST LABEL VERSION

We found that this algorithm accumulates flow near the sink before pushing it into the sink. The last pushes did the work of pushing all the flow into the sink and no relabels were performed once the flow began reaching the sink.

For this algorithm also, we observe in Figures 4.32 and 4.33 that the number of pushes required to convert the maximum preflow to a maximum flow using a modification of the highest label version of preflow-push algorithm, is very less. They are less than 0.5n and comprise less than 5% of the total pushes performed. There were very few node relabels during the second phase.

(c) WAVE VERSION

We observe that the wave version of the preflow-push algorithm, which is a hybrid of the FIFO and the highest label versions, has an empirical performance better than the FIFO version and worse than the highest label version of the preflow-push algorithm. In Figure 4.57, we can observe that the wave version improves by about 10%-15% on the FIFO version regarding CPU time, non-saturating pushes, and arc scans for relabels.

4.6 A COMPARISON OF SOME PREFLOW-PUSH ALGORITHMS

In this section, we compare Karzanov's algorithm, the FIFO and highest level versions of the preflow-push algorithm and the stack scaling algorithm (to be described in Chapter 5) .

From Figure 4.58 through Figure 4.61 which are representative curves, we can observe that the highest label version of the preflow-push algorithm performs the maximum number of saturating pushes and the minimum number of non-saturating pushes among the algorithms being compared. Karzanov's algorithm performs the smallest number of saturating pushes and the largest number of non-saturating pushes. The total pushes required to establish a maximum flow is least for the highest label implementation and largest for the Karzanov's algorithm [see Figures 4.62 and 4.63]. We can see in Figures 4.64 and 4.65 that the Karzanov's algorithm saturates the arcs in only about 10% to 25% of the total pushes. For the highest label implementation of the preflow-push algorithm, the saturating pushes comprise about 40% to 50% of the total pushes and for the FIFO version they are about 30% to 40% of the total pushes. It can be seen in these figures that Karzanov's algorithm performs more than 5 times the total pushes performed by the FIFO and the highest level versions.

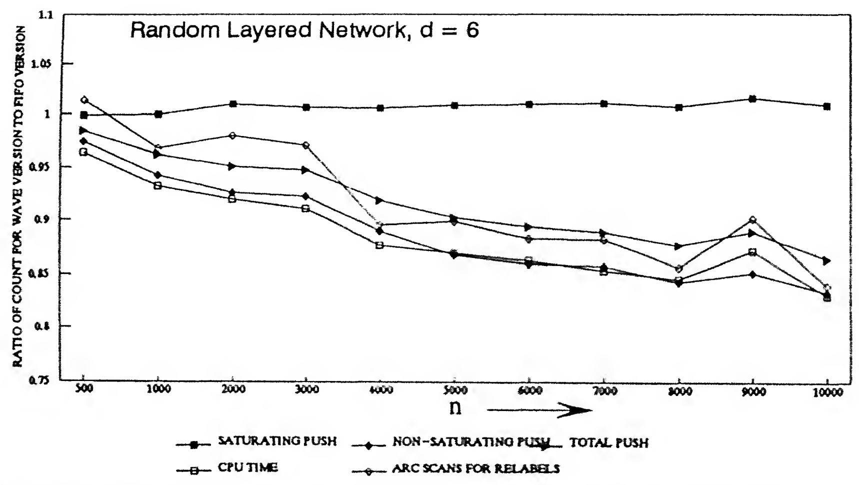


Figure 4.57. Comparison of Wave and FIFO versions of preflow–push algorithm

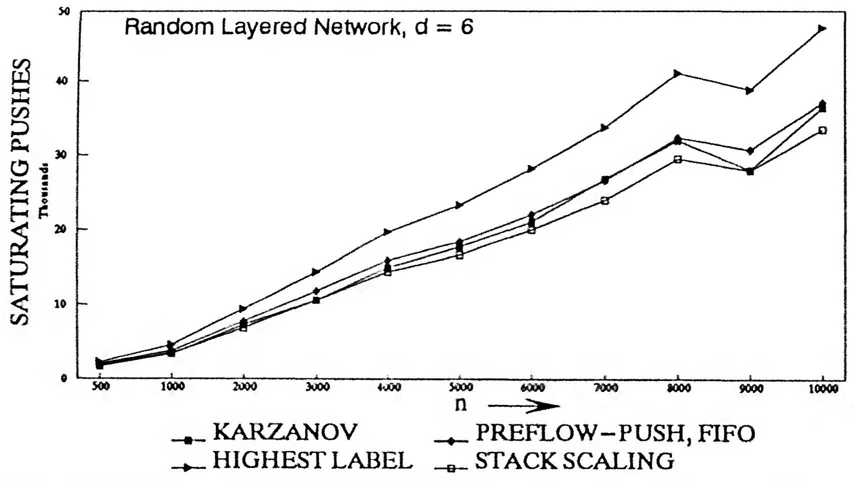


Figure 4.58. Saturating pushes on random layered network

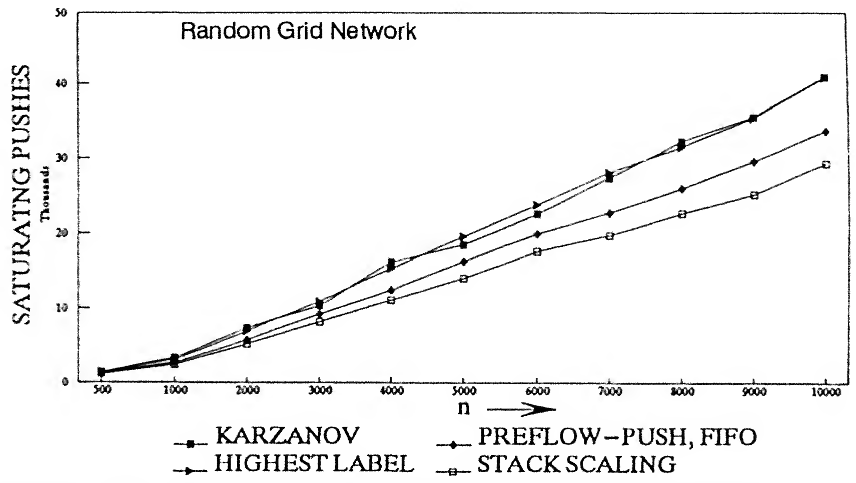


Figure 4.59. Saturatig pushes on random grid network

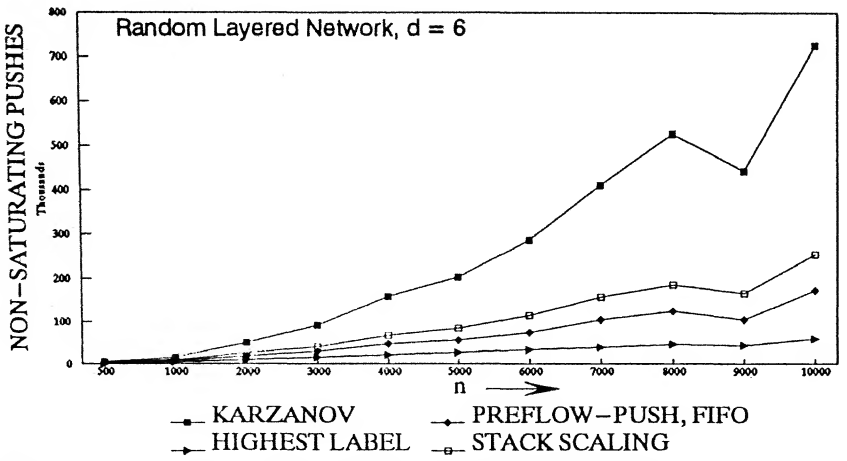


Figure 4.60. Non-saturating pushes on random layered network

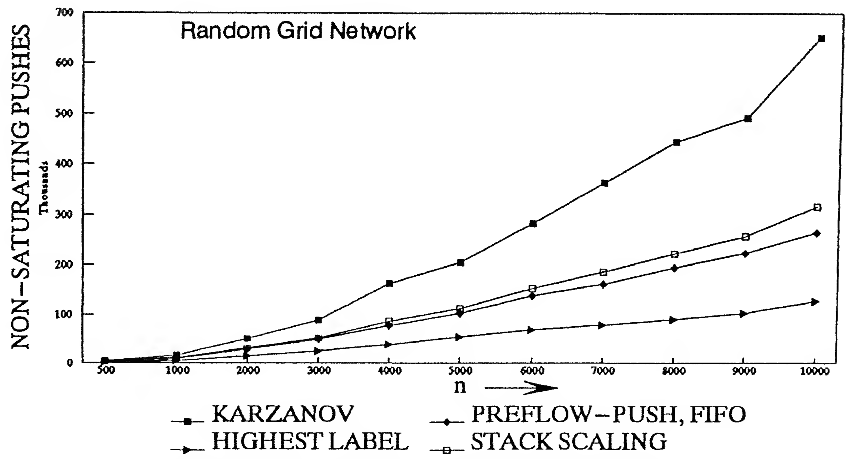


Figure 4.61. Non-saturating pushes on random grid network

COMPARISON OF SOME PREFLOW-PUSH ALGORITHMS

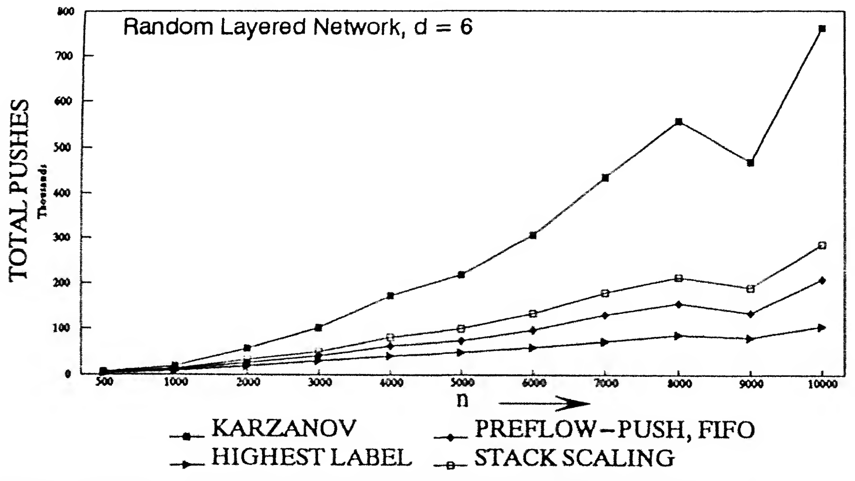


Figure 4.62. Total pushes on random layered network

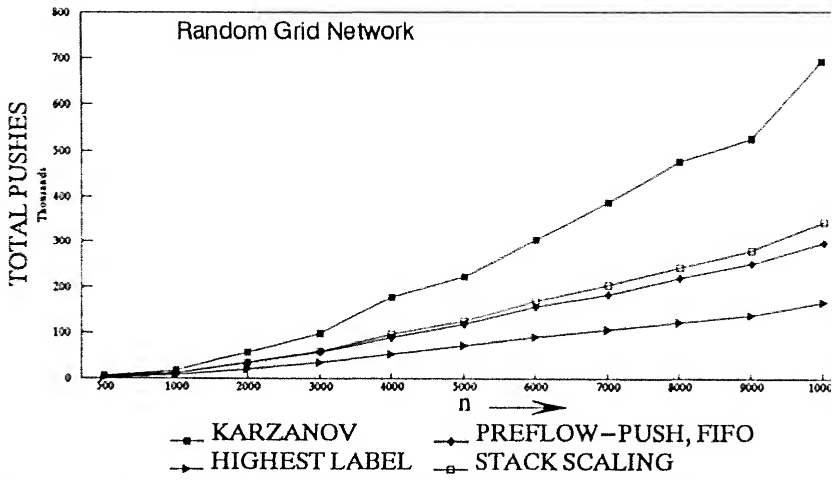


Figure 4.63. Total pushes on random grid network

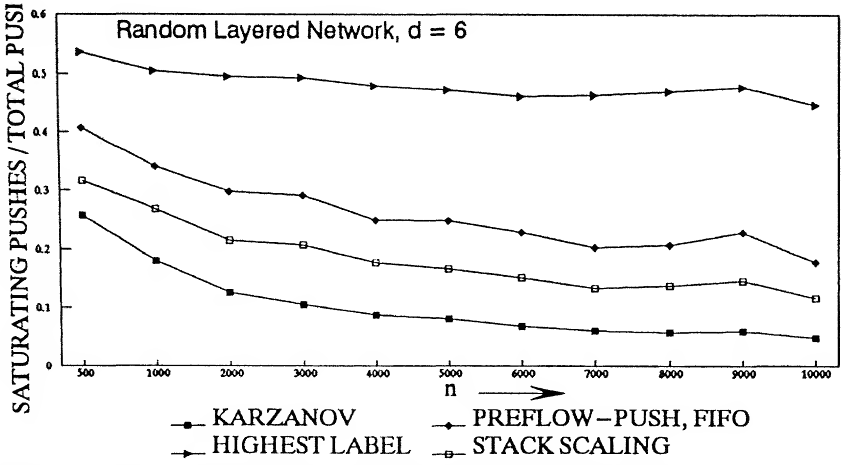


Figure 4.64. Proportion of saturating pushes in total pushes on random layered network

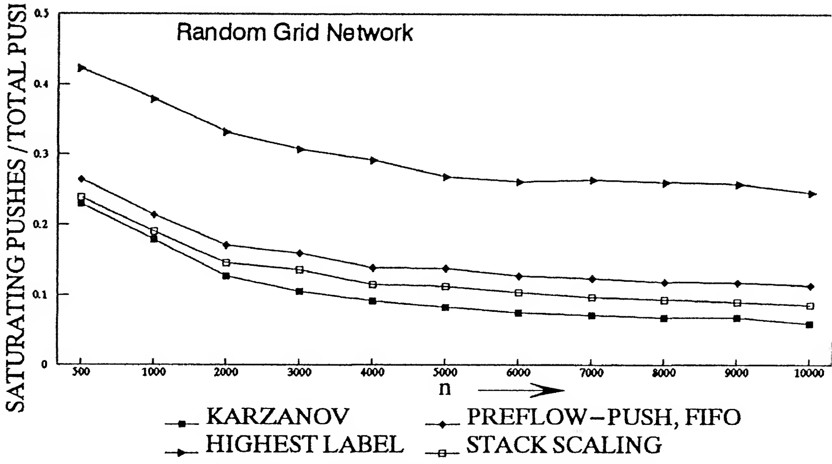


Figure 4.65. Proportion of saturating pushes in total pushes on random grid network

COMPARISON OF SOME PREFLOW-PUSH ALGORITHMS

The total number of arc scans in Karzanov's algorithm to construct layered networks is about 10 to 20 times the relabeling time in the highest label version of the preflow-push algorithm [see Figures 4.2 , 4.5, 4.66 and 4.67]. The FIFO version performs about 2-3 times more arc scans for relabels than the highest label version.

A comparison of the CPU times taken by these algorithms in Figures 4.68 and 4.69, shows that the highest label implementation of the preflow-push algorithm is about 5 to 10 times faster than Karzanov's algorithm on both the types of networks and 2 to 3 times faster than the FIFO version.

So, we observe that the highest label version of the preflow-push algorithm is superior to all other preflow-push algorithms.

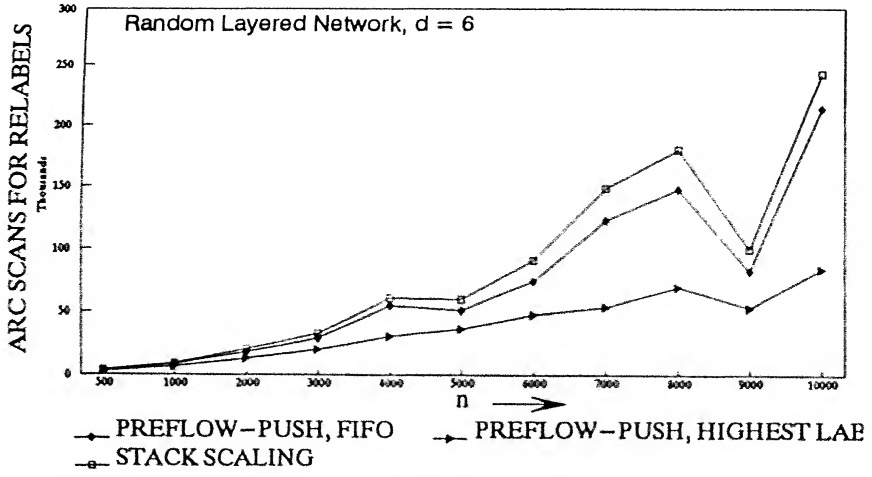


Figure 4.66. Arc scans for relabels on random layered network

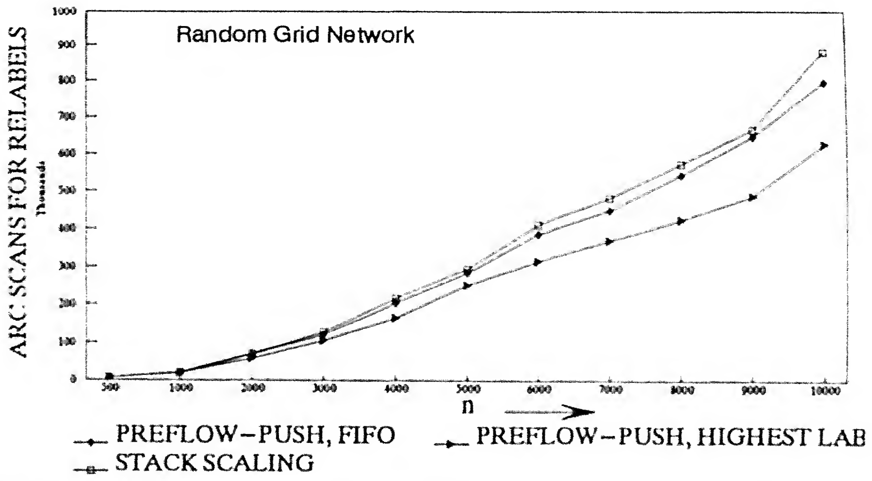


Figure 4.67. Arc scans for relabels on random grid network

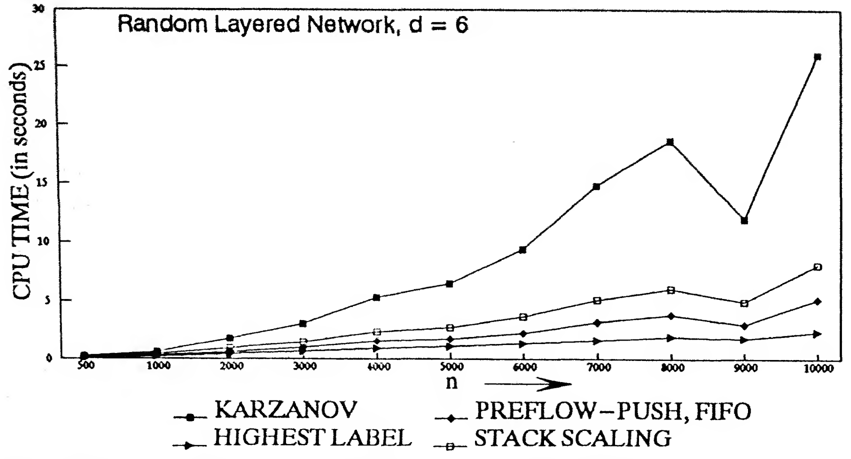


Figure 4.68. CPU time taken on random layered network

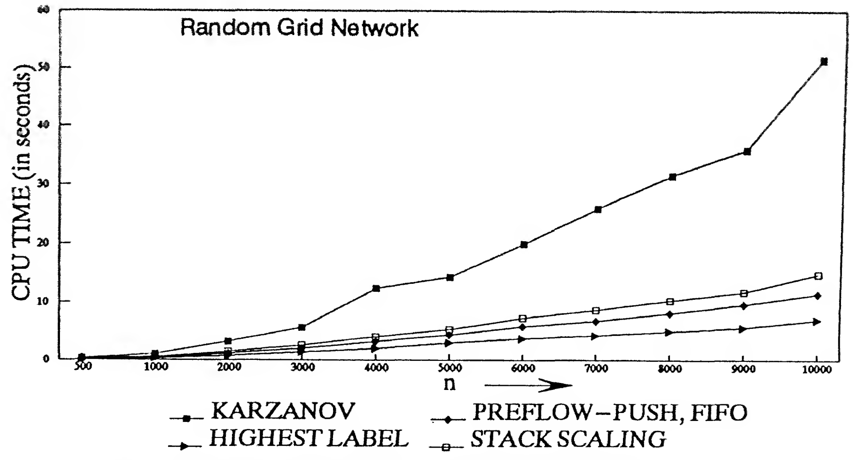


Figure 4.69. CPU time taken on random grid network

COMPARISON OF SOME PREFLOW–PUSH ALGORITHMS

CHAPTER 5

EXCESS SCALING ALGORITHMS

5.1 INTRODUCTION

Excess scaling algorithms are special implementations of the generic preflow push algorithm and using a scaling technique dramatically improve the number of non-saturating pushes in the worst case. The basic idea is to push flow from active node with sufficiently large excesses to nodes with sufficiently small excesses while never letting the excesses become too large. For problems, that satisfy the similarity assumption, these algorithms have the worst-case time bound better than the preflow-push algorithms discussed in the previous chapter.

5.2 EXCESS SCALING ALGORITHMS

The essential idea in the excess scaling algorithms is to assure that each non-saturating push carries "sufficiently large" flow so that the number of non-saturating pushes is "sufficiently small". The algorithm defines the term "sufficiently large" iteratively.

Let $e_{max} = \max\{e(i): i \text{ active}\}$ and Δ be an upper bound e_{max}; we refer to this bound as the *excess-denominator*. We call a node with $e(i) \geq \Delta/2 \geq e_{max}/2$ as a node with *large excess,* and a node with *small excess* otherwise. The excess scaling algorithm always pushes flow from a node with large excess. This choice assures that during non-saturating pushes, the algorithm sends relatively large excess closer to the sink. The excess scaling algorithm also does not allow the maximum excess to increase beyond Δ. These two conditions, namely, that each non-saturating push must carry at least $\Delta/2$ units of flow and that no excess should exceed Δ, imply that we have to select active nodes for *push/relabel* operations carefully. One selection rule that assures this is as follows: *Among all nodes with large excess, select a node with smallest distance label (breaking ties arbitrarily).* A formal description of the excess scaling algorithm is given below.

```
algorithm excess-scaling;
begin
    pre-process;
    $\Delta$: $2^{\lceil \log U \rceil}$;
    while $\Delta \geq 1$ do;
    begin $\Delta$-scaling phase;
      while the network contains a node i with large excess do
      begin
        among all nodes with large excess, select a node i with smallest
            distance label;
        perform push/relabel(i) while ensuring that no node excess
            exceeds $\Delta$;
      end;
      $\Delta := \Delta/2$
    end;
end;
```

The excess-scaling algorithm uses the same *push/relabel(i)* step as the generic preflow push algorithm but with one slight difference. Instead of pushing $\delta = \min\{e(i), r_{ij}\}$units of flow, it pushes $\delta = \min\{e(i), r_{ij}, \Delta - e(j)\}$ units. This change ensures that the algorithm permits no excess to exceed Δ.

The algorithm performs a number of scaling phases with the value of the excess denominator Δ decreasing from phase to phase. In the last scaling phase $\Delta = 1$ and hence the preflow at the end of this phase is a flow. This establishes the correctness of the excess scaling algorithm. We now briefly discuss the worst case complexity of the algorithm.

Lemma 5.1. The algorithm satisfies the following two conditions:

(i) Each non-saturating push sends at least $\Delta/2$ units of flow

(ii) No excess ever exceeds Δ.

Proof. Omitted. ■

Lemma 5.2. The excess scaling algorithm performs $O(n^2 \log U)$ non-saturating pushes.

Proof Sketch. We use the potential function $F = \sum_{i \in N} e(i)\ d(i)/\Delta$ to prove the lemma.

The potential function increases whenever the scaling phases begins and Δ is replaced by $\Delta/2$. This increase is $O(n^2)$ per scaling phase and $O(n^2 \log U)$ over all scaling phases. Increasing the distance label of a node i by E increases F by at most E (because

$e(i) \le \Delta$). Thus the total increase in F due to relabeling is $O(n^2)$ over all scaling phases (by Lemma5.1). Each non-saturating push carries at least $\Delta/2$ units to flow to a node with smaller distance label and hence decrease F by at least $\Delta/2$ units. These observations give a bound of $O(n^2 \log U)$ on the number of non-saturating pushes. ■

The above lemma implies that the above algorithm runs in $O(nm + n^2 \log U)$ time as all other operations require a total of O(nm) time.

5.3 STACK SCALING ALGORITHM

The stack scaling algorithm scales excesses by a factor of k and always pushes flow from a large excess node (a node with excess of at least Δ/k) with the highest distance label. Observe that since the algorithm pushes $\min\{e(i), r_{ij}, \Delta - e(j)\}$ units while performing pushes on the arc (i, j), it may not be able to push at least $\Delta/2$ units of flow in a non-saturating push if node j is also a large excess node. To overcome this difficulty, the stack scaling algorithm performs a sequence of push and relabel steps using a stack S. Suppose we want to examine a large excess node i until either node i becomes a small excess node or node i relabeled. Then we set $S = \{i\}$ and repeat the following steps until S is empty.

Stack Push: Let v be the top node on S. Identify an admissible arc out of v. If there is no admissible arc, then relabel node v and pop (delete) v from S. Otherwise, let (v, w) be an admissible arc. There are two cases.

Case 1. $e(w) > \Delta/2$ and $w \neq t$. Push w onto S.

Case 2. $e(w) \le \Delta/2$ or $w = t$. Push $\min\{e(v), r_{ij}, \Delta - e(w)\}$ units of flow on arc (v, w). If $e(v) \le \Delta/2$, then pop node v from S.

It can be shown that if we choose $k = \lceil \log U/\log \log U \rceil$ then the stack scaling algorithm performs $O(n^2 \log U/\log \log U)$ non-saturating pushes and runs in $O(nm + n^2 \log U/\log \log U)$ time.

5.4 WAVE SCALING ALGORITHM

The wave scaling algorithm scales excesses by a factor of 2 and uses a parameter L whose value is chosen appropriately. This algorithm differs from the excess scaling algorithm as follows. At the beginning of every scaling phase, the algorithm checks whether $\sum_{i \in N} e(i) > n\Delta/L$). If yes, then we run the wave algorithm described in Section 4.3(c) on the active nodes. The algorithm examines all active nodes in the non-increasing order of their distance labels and performs pushes at each such node until either its excess reduces to zero or the node is relabeled. Note that if the wave algorithm is applied as such then excesses at nodes may exceed Δ which is not allowed. Therefore we apply the wave algorithm using *stack pushes* as described in the stack scaling algorithm. We terminate the wave algorithm when we find that $\sum_{i \in N} e(i) \geq n\Delta/L)$. At this point, we apply the original excess scaling algorithm, i.e., we push flow from a large excess node (having excess at least) with the smallest distance label. Ahuja, Orlin and Tarjan [1989] showed that by choosing $L = \sqrt{\log U}$, the algorithm can be shown to perform $O(n^2 \sqrt{\log U})$ non-saturating pushes and run in $O(nm + n^2 \sqrt{\log U})$ time.

5.5 COMPUTATIONAL RESULTS FOR EXCESS SCALING ALGORITHM

The representative operations for this algorithm are the pushes and arc scans for relabels. We count these and the saturating and non-saturating pushes and relabel operations.

Figures 5.1 and 5.2 show the growth of pushes and arc scans for relabels for the random layered network. Figure 5.5 shows their increase with the increase in size of the random grid network. From Figures 5.3 and 5.4 we observe that the arc scans for relabels have an increasing share in the sum of the representative operation counts but for random layered networks of size upto 10,000 nodes the pushes dominate the arc scans for relabels. From Figure 5.5, we see that on the random grid network, the arc scans for relabels is not the bottleneck operation for n=500 and for larger networks the arc scans for relabels dominate the pushes.

In Figure 5.7, we see that the saturating pushes comprise only 5 to 20% of the total pushes. Figure 5.8 shows the CPU time taken by the excess scaling algorithm.

We try to estimate the CPU time using the representative operation counts. We obtained the virtual running time for this algorithm as a function of the total pushes and the arc scans for relabels.

$$V(I) = C_P \text{ (total pushes)} + C_R \text{ (arc scans for relabels)}.$$

where C_P and C_R are constants. We use multiple regression analysis to estimate C_P and C_R for networks of the same density but different sizes.

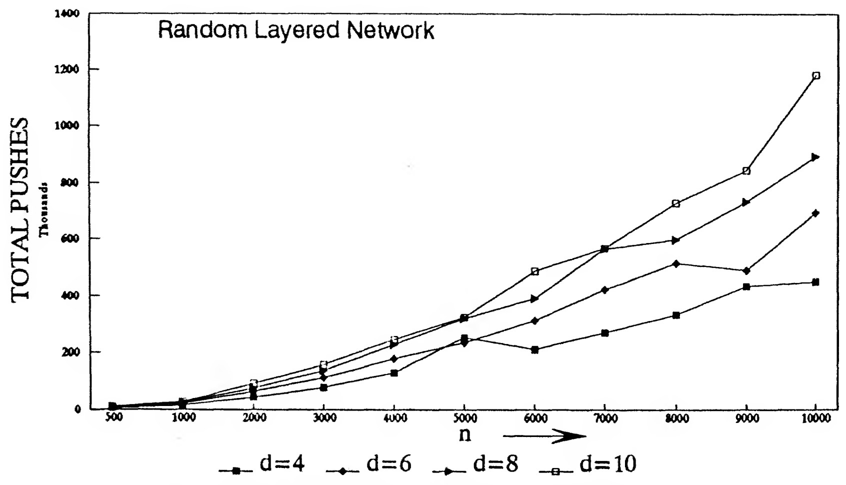


Figure 5.1. Representative operation: total pushes

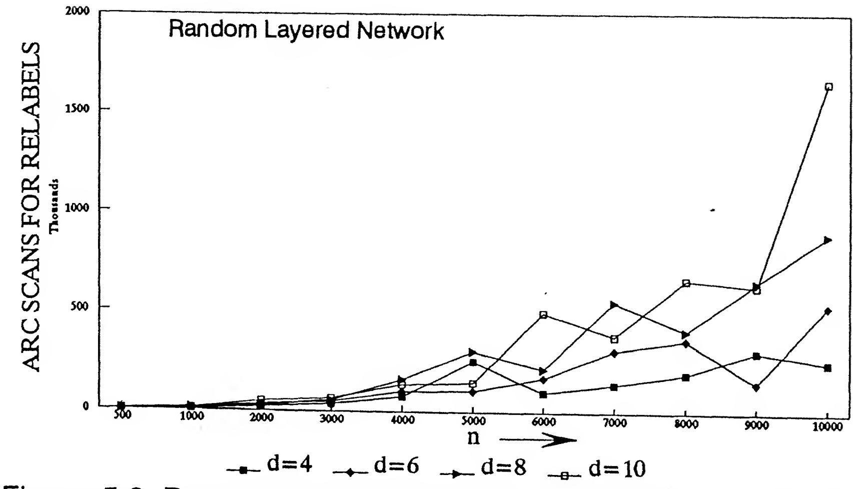


Figure 5.2. Representative operation: arc scans for relabels

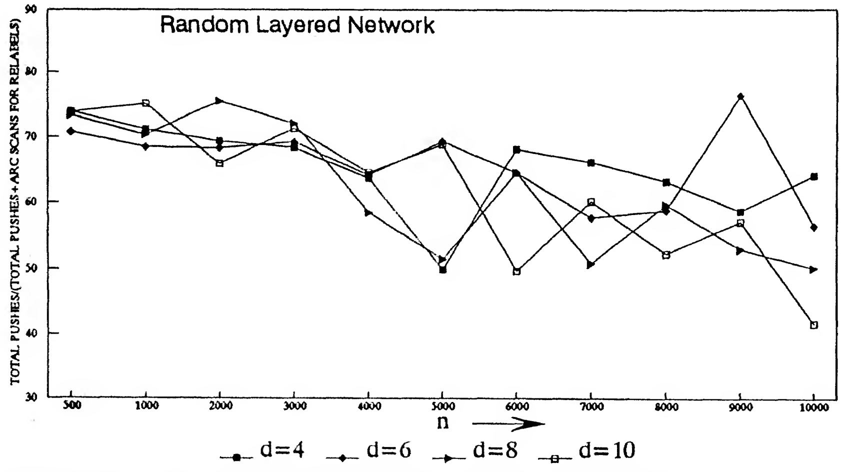


Figure 5.3. Identifying asymptotic bottleneck operation: Ratio of total pushes to total representative opn. counts

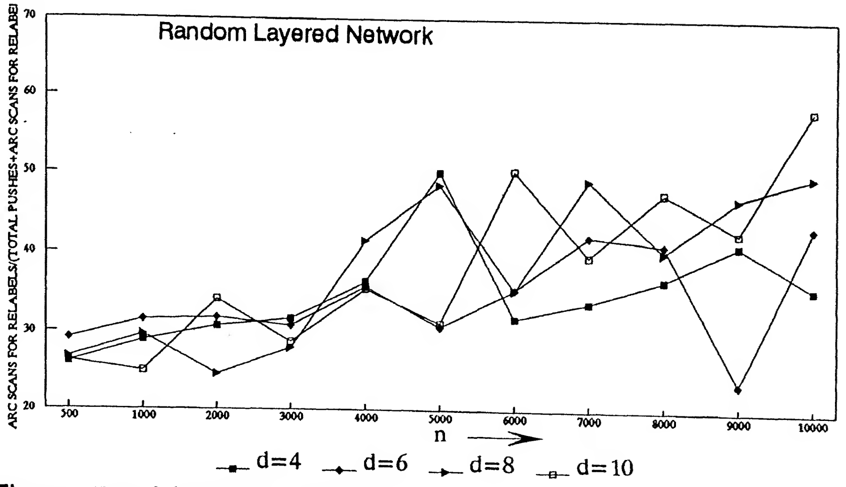
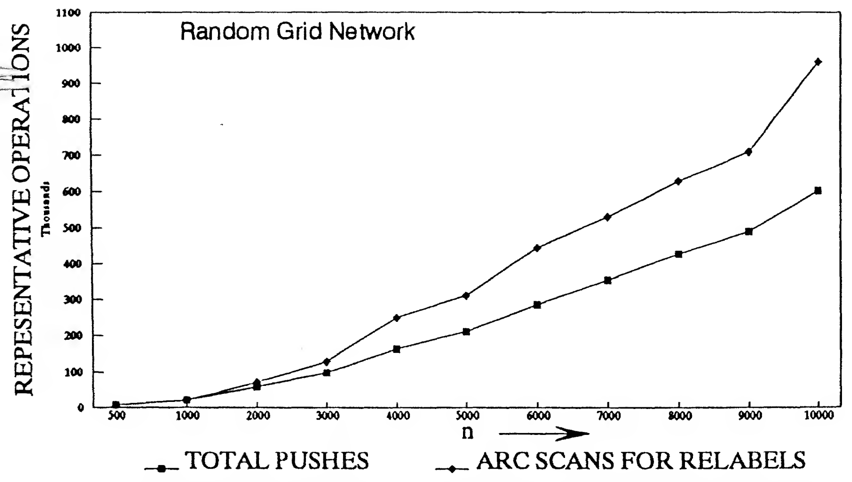


Figure 5.4. Identifying asymptotic bottleneck operation:Ratio of arc scans for relabels to total representative opn. counts

EXCESS SCALING ALGORITHM



Figue 5.5. Representative operations on random grid network

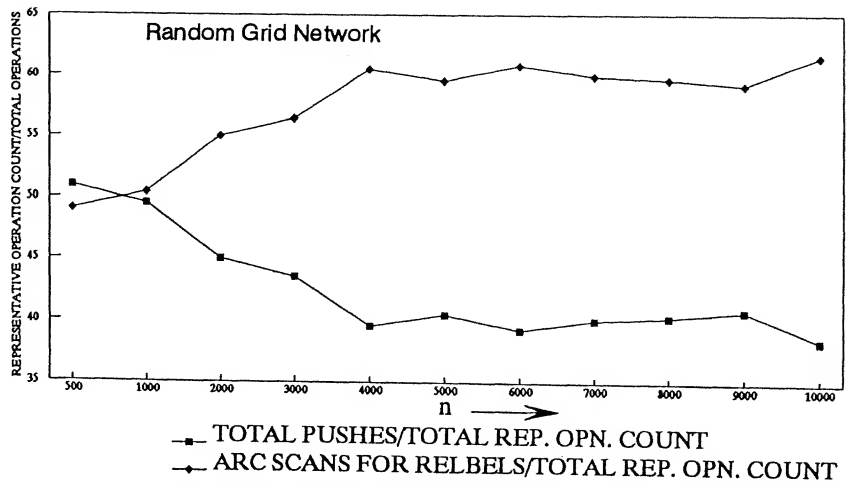


Figure 5.6. Identifying asymptotic bottleneck operation: ratio of representative operation count to total rep. opn. count

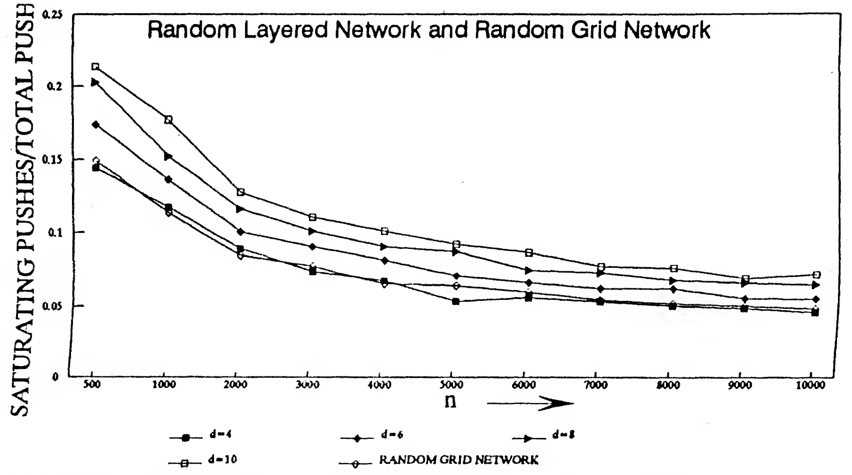


Figure 5.7. Proportion of saturating pushes in total pushes

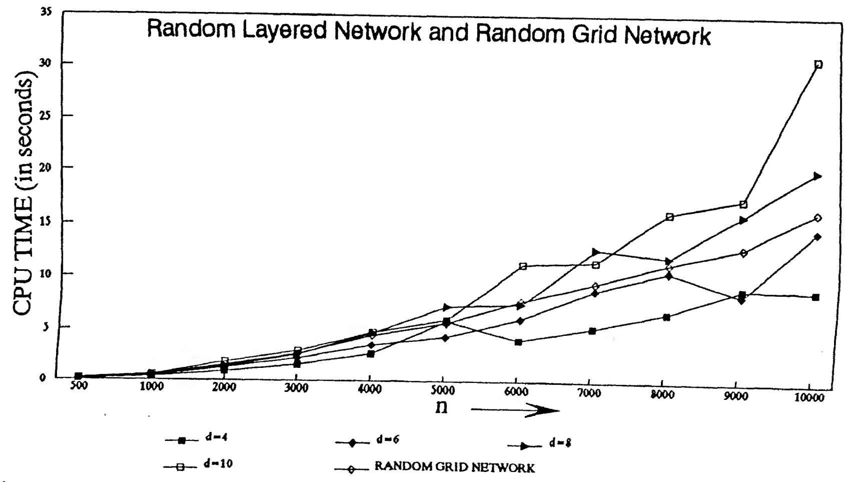


Figure 5.8. CPU time taken

EXCESS SCALING ALGORITHM

Network	Density (d)	C_P (10^{-6})	C_R (10^{-6})
Random Layered Network	4	15.5	7.7
	6	14.9	7.9
	8	14.7	8.1
	10	14.7	8.2
Random Grid Network		19.3	5.0

Table 5.1. Regression coefficients in virtual running time estimation for excess scaling algorithm.

We observe in Figure 5.9 that the virtual running time does not approximate the CPU time well for networks of sizes less than n= 2,000. As seen in Figure 5.10 and 5.11, the pushes take a large share of the virtual running time for $n \leq 10{,}000$ nodes and hence the arc scans for relabels are not the bottleneck operation for networks of this size.

We observe that the total pushes grow at about $n^{1.4}$ and $n^{1.5}$. The non-saturating pushes also vary between $n^{1.4}$ and $n^{1.5.}$ The arc scans for relabels vary as g and the growth rate of relabels is between n and $n^{1.25}$ [see Figures 5.12 through 5.15].

We changed the maximum capacity U of an arc on a network and plotted the number of non-saturating pushes since there are (n^2 logU) non-saturating pushes. We observe that changing the value of U does not appreciably change the number of non-saturating pushes. [see Figure 5.16].

The excess scaling algorithm incorporates scaling in the lowest label preflow-push algorithm. The lowest label version has an unattractive performance. The excess scaling algorithm reduces the non-saturating pushes by more than 4 times. This is shown for representative problems on the random layered network and the random grid network in Figure 5.17.

5.6 COMPUTATIONAL RESULTS FOR STACK SCALING ALGORITHM

Figures 5.18 through 5.23 show the variation of the pushes and the arc scans for relabels for different size networks and relative to each other. The arc scans for relabels is the asymptotic bottleneck operation for this algorithm as is evident from these figures. Figure 5.24 shows the proportion of the saturating pushes in the total pushes. The saturating pushes comprise about 10 to 40% of the total pushes.

We obtained the virtual running time for this algorithm as a function of the total pushes and the arc scans for relabels.

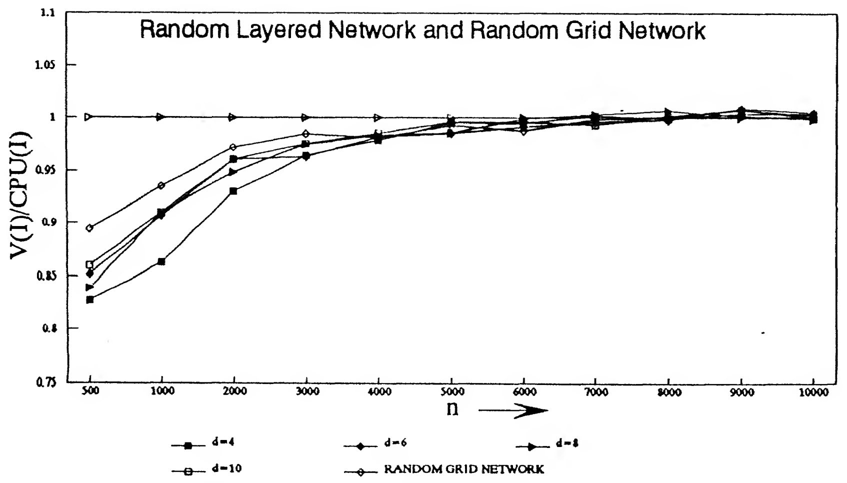


Figure 5.9. Approximation of CPU time by virtual running time

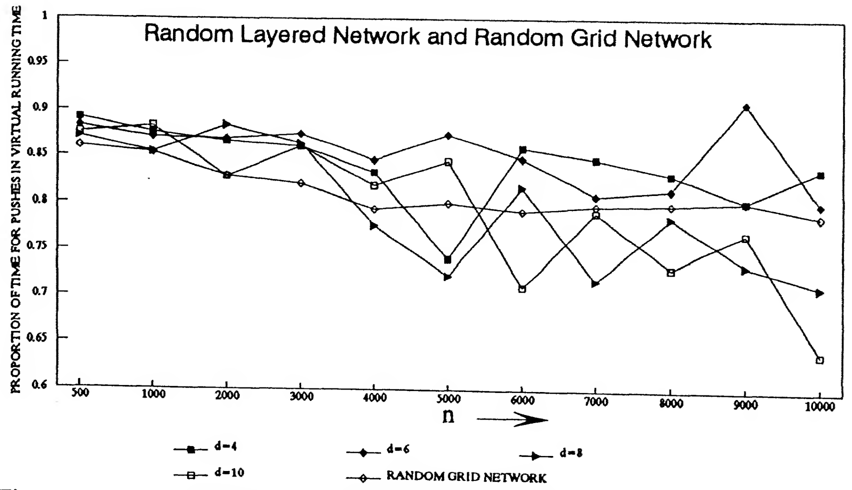


Figure 5.10. Proportion of time for pushes in virtual running time

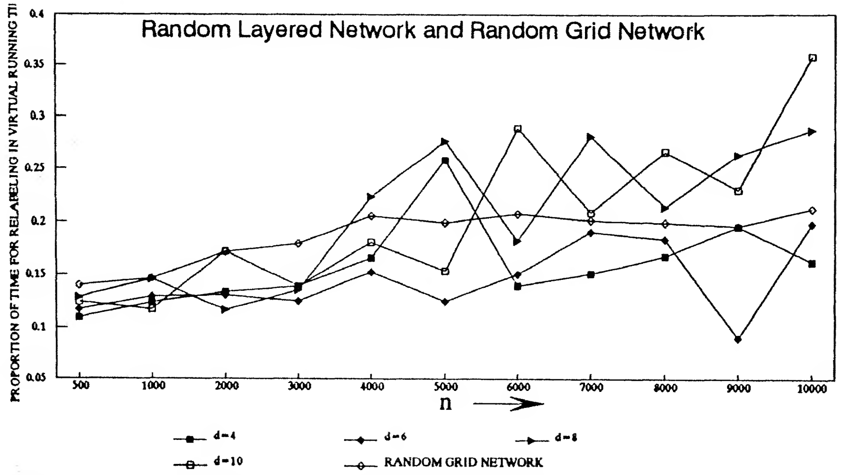


Figure 5.11. Proportion of relabeling time in virtual running time

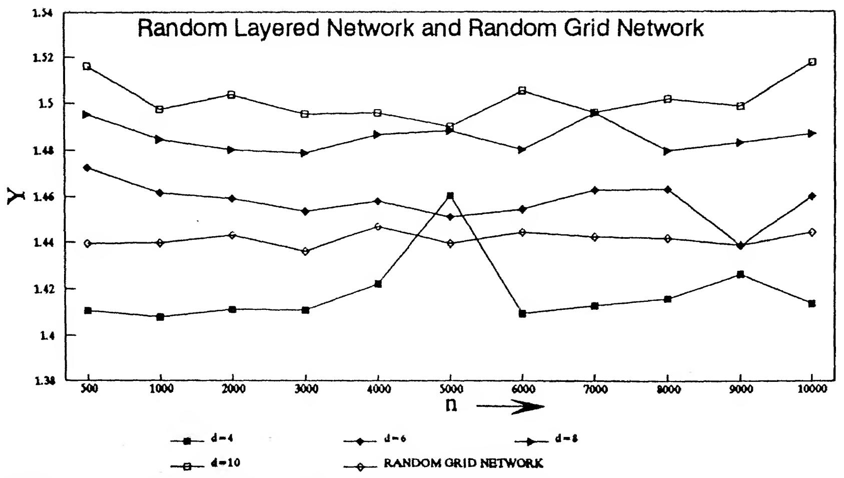


Figure 5.12. Growth rate of total pushes
Y = log(total pushes)/log(n)

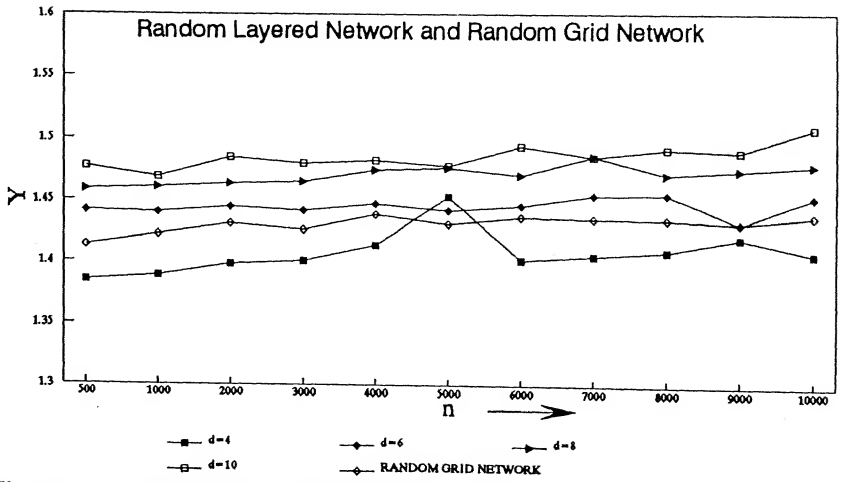


Figure 5.13. Growth rate of non–saturating pushes
Y = log(non–saturating pushes)/log(n)

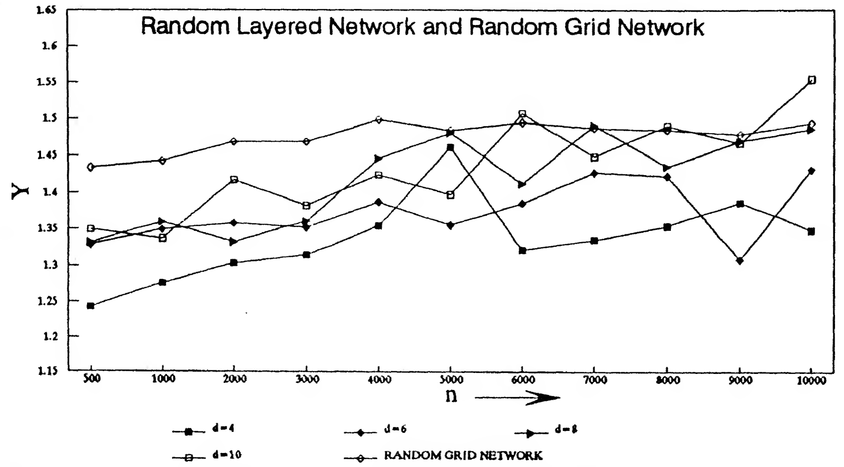


Figure 5.14. Growth rate of arc scans for relabels
Y = log(arc scans for relabels)/log(n)

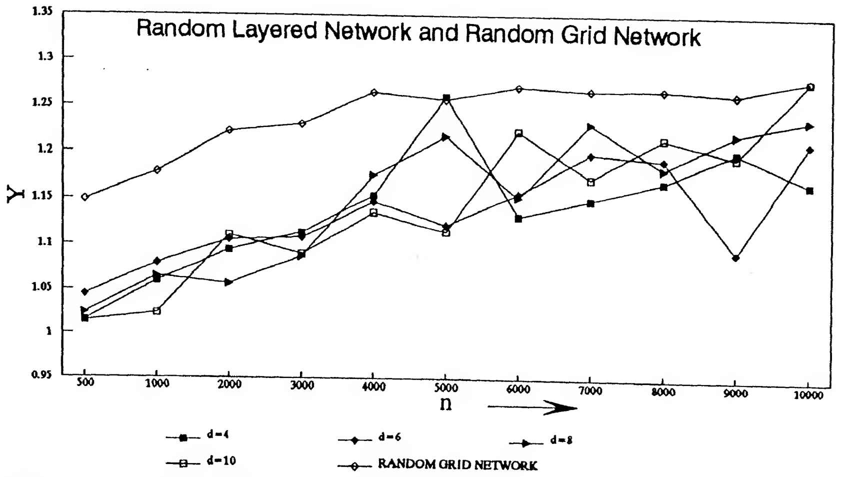


Figure 5.15. Growth rate of relabels
Y = log(# of relabels)/log(n)

EXCESS SCALING ALGORITHM

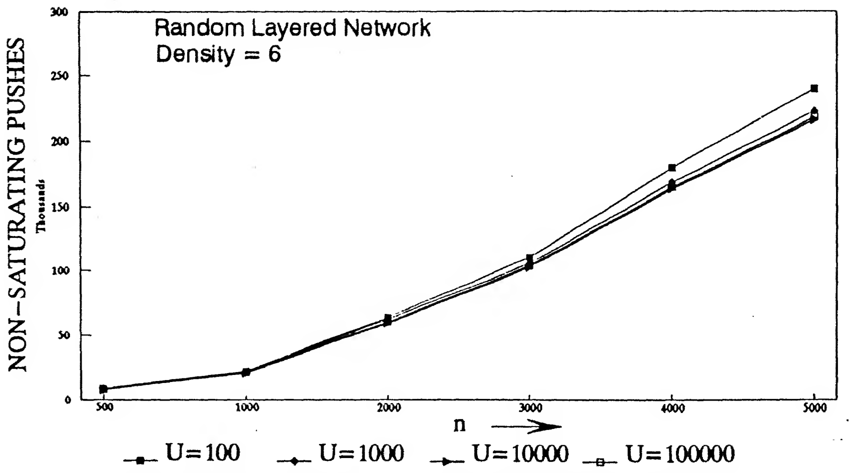


Figure 5.16. Effect of varying upper bound on arc capacity, U on the non-saturating pushes

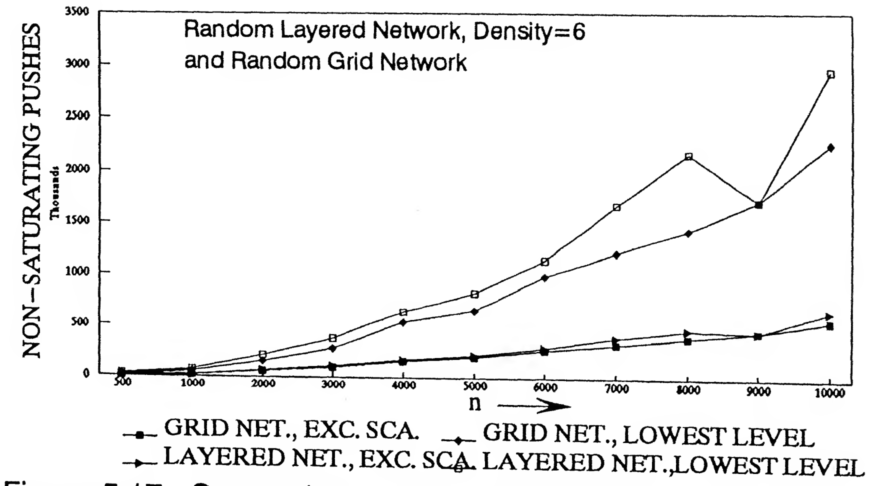


Figure 5.17. Comparison of the number of non-saturating pushes performed by excess scaling algorithm and lowest level version of preflow-push algorithm

EXCESS SCALING ALGORITHM

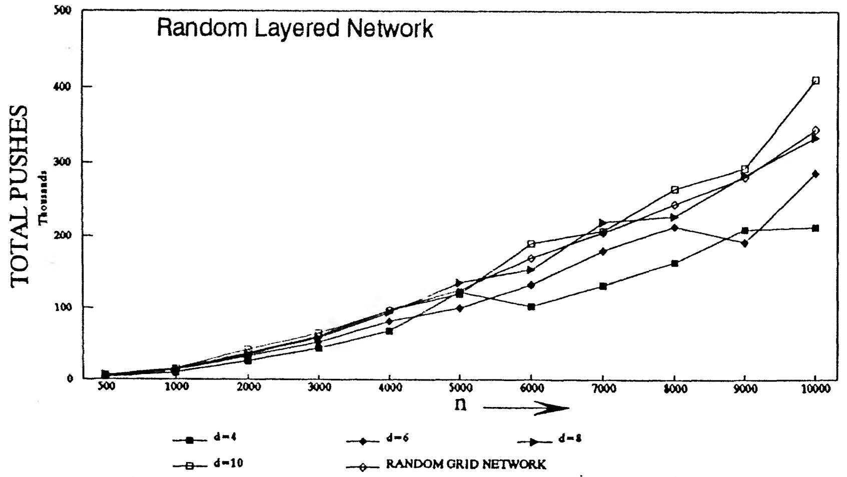


Figure 5.18. Representative operation: total pushes

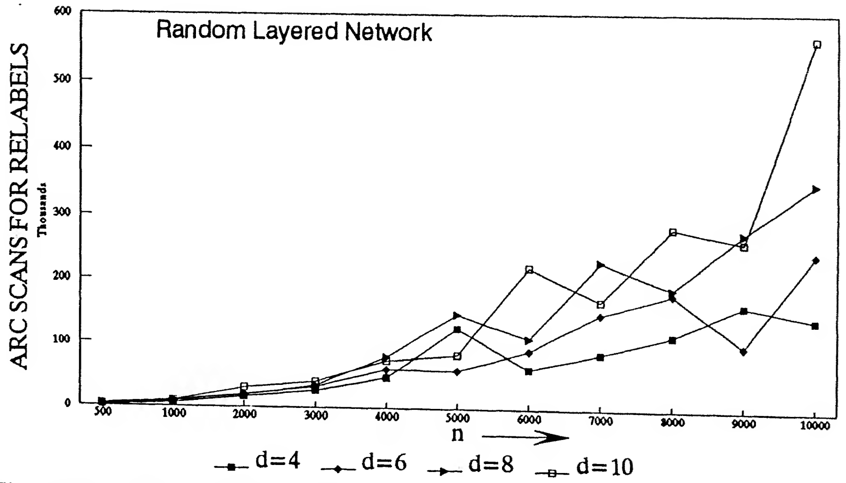


Figure 5.19. Representative operation: arc scans for relabels

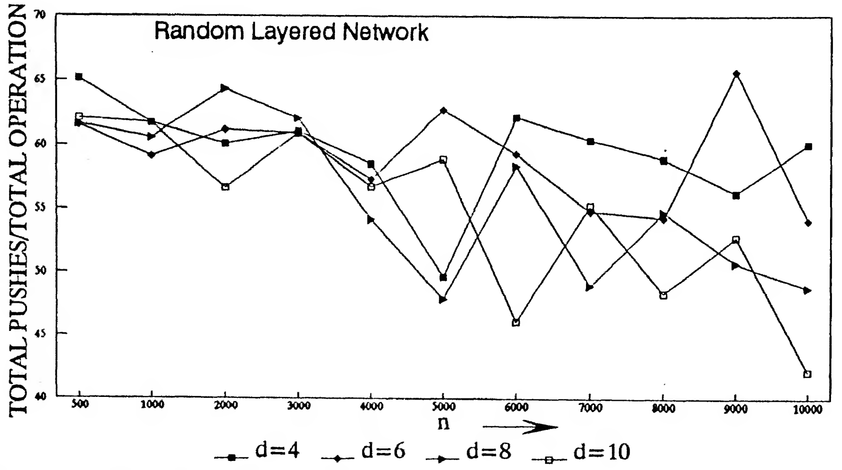


Figure 5.20. Identifying the asymptotic bottleneck operation: ratio of total pushes to total representative opn. counts

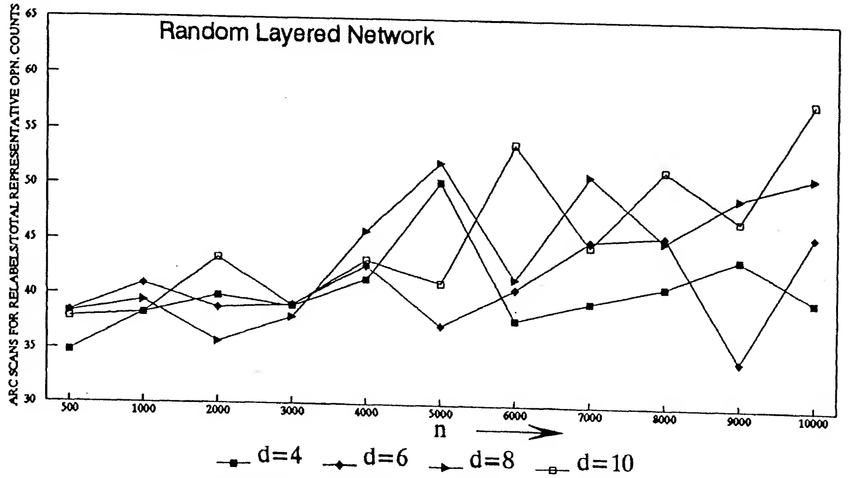


Figure 5.21. Identifying asymptotic bottleneck operation: ratio of arc scans for relabels to total representative opn. counts

STACK SCALING ALGORITHM

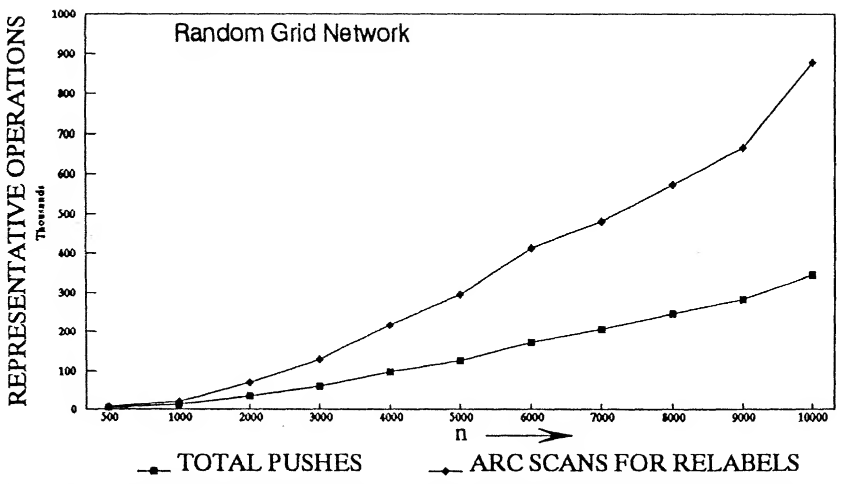


Figure 5.22. Representative operations: total pushes and arc scans for relabels

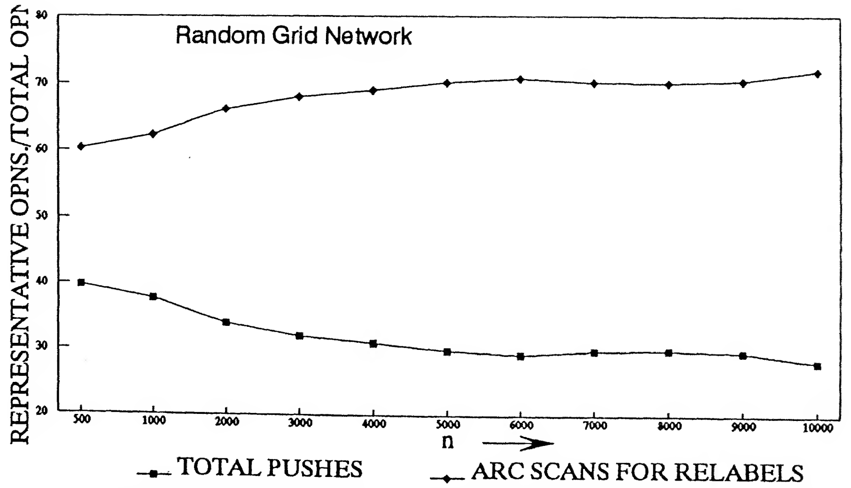


Figure 5.23. Identifying asymptotic bottleneck operation:ratio of representative operation count to total rep. opns. count

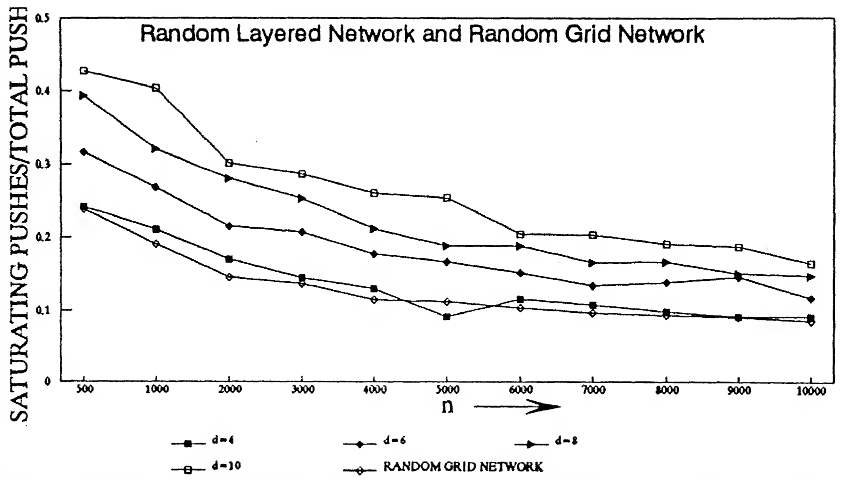


Figure 5.24. Proportion of saturating pushes in total pushes

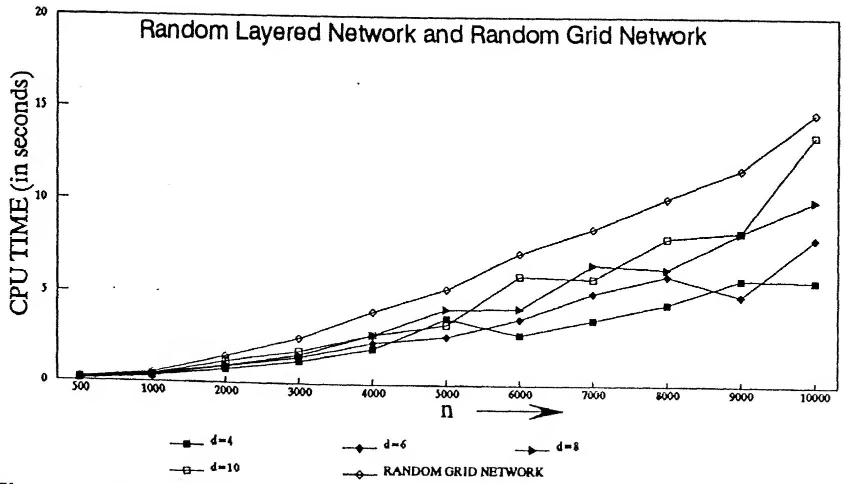


Figure 5.25. CPU time taken

STACK SCALING ALGORITHM

where C_P and C_R are constants. We use multiple regression analysis to estimate C_P and C_R for networks of the same density but diferent sizes.

Network	Density (d)	C_P (10^{-6})	C_R (10^{-6})
Random Layered Network	4	22	7.7
	6	22.2	7.2
	8	21.3	8.6
	10	21.3	8.6
Random Grid Network		30.1	5.05

Table 5.2. Regression coefficients in virtual running time estimation for stack scaling algorithm.

In Figure 5.26, we find that the virtual running time is not a good approximation for the CPU time for networks of smaller sizes. A study of the percentage of the virtual running time accounted by the pushes (see Figure 5.27]) shows that the pushes account for 75% to 85% of the virtual running time and hence are the bottleneck operation for $n \leq 10{,}000$. The relabeling time is not a bottleneck operation for the algorithm on networks of the size we used in this study.

An effort to estimate the growth rate of the operations as a function of n^γ is made. We observe from Figures 5.29 through 5.32 that the pushes vary between $n^{1.35}$ and $n^{1.4}$. The non-saturating pushes vary between $n^{1.3}$ and $n^{1.35}$. The growth rate of arc scans for relabels is between $n^{1.25}$ and $n^{1.32}$ and the relabels vary between n and $n^{1.25}$.

5.7 COMPUTATIONAL RESULTS FOR WAVE SCALING ALGORITHM

We performed the tests on the wave scaling algorithm and the curves are shown in Figures 5.33 through 5.47. The arc scans for relabels is the asymptotic bottleneck operation for the wave scaling algorithm as seen in Figures 5.35 and 5.36 for the random layered network and Figure 5.38 for the random grid network.

Figure 5.39 shows that the saturating pushes have a share of 10% to 35% in the total pushes. The CPU time taken by the algorithm for different problems is shown in Figure 5.40.

The virtual running time is found using the repesentative operations :total pushes and arc scans for relables.

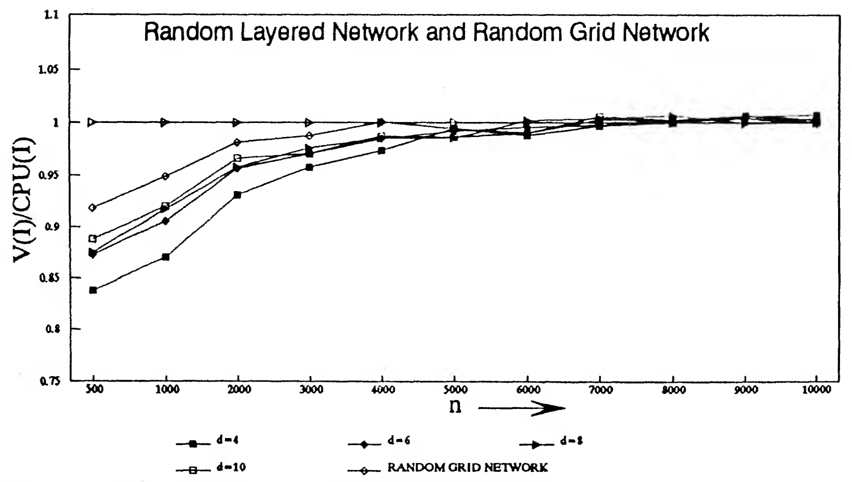


Figure 5.26. Approximation of CPU time by virtual running time

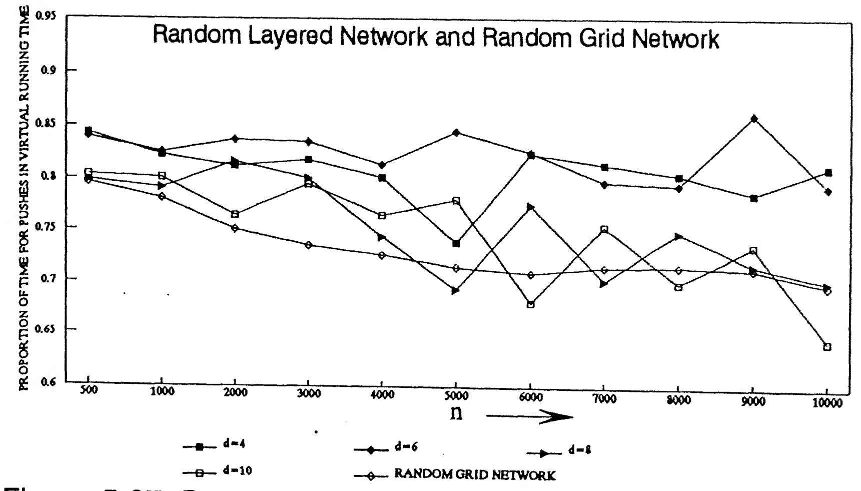


Figure 5.27. Proportion of time for pushes in virtual running time

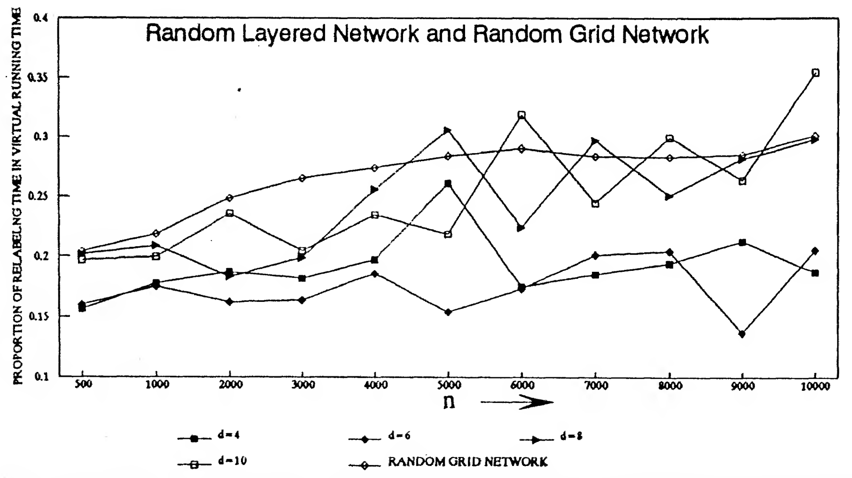


Figure 5.28. Proportion of relabeling time in virtual running time

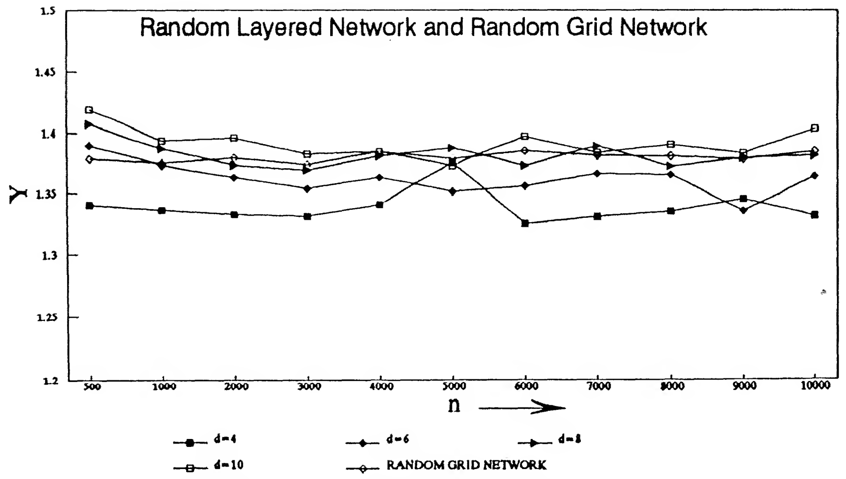


Figure 5.29. Growth rate of total pushes
Y = log(total pushes) / log(n)

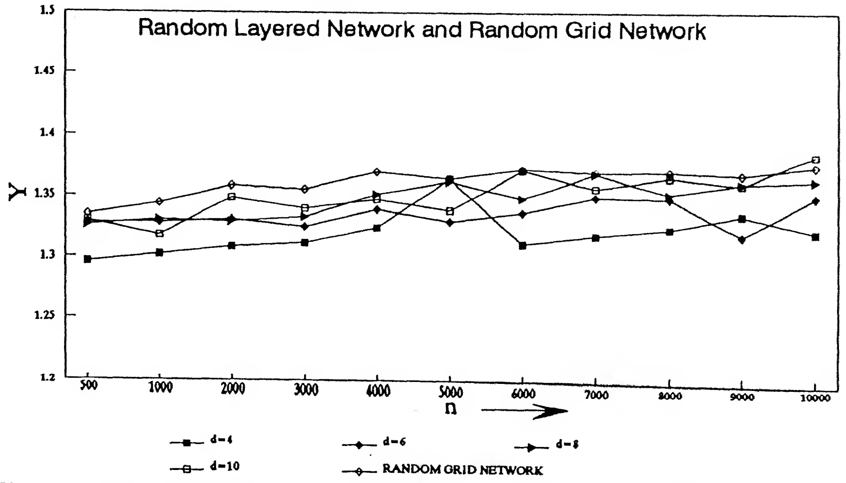


Figure 5.30. Growth rate of non−saturating pushes
Y = log(non−saturating pushes) / log(n)

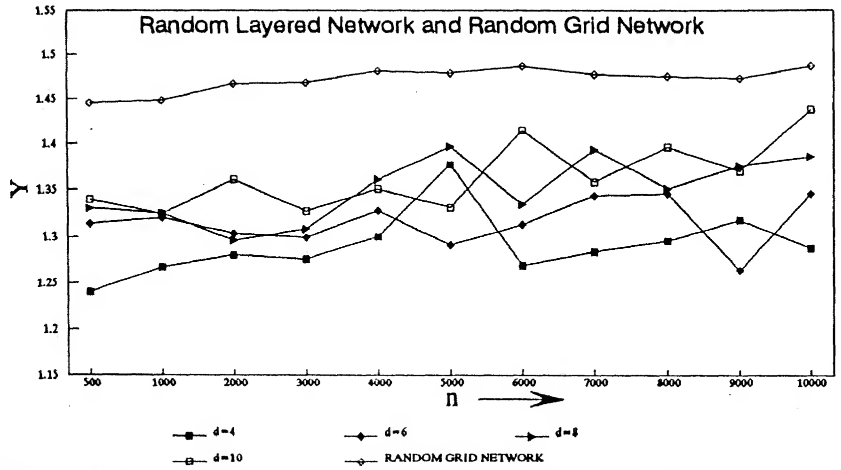


Figure 5.31. Growth rate of arc scans for relabels
Y = log(arc scans for relabels) / log(n)

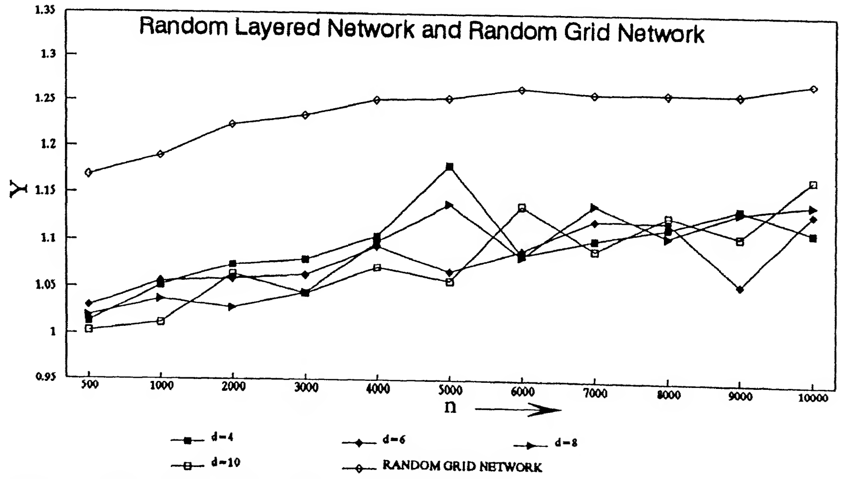


Figure 5.32. Growth rate of relabels
Y = log(# of relabels) / log(n)

STACK SCALING ALGORITHM

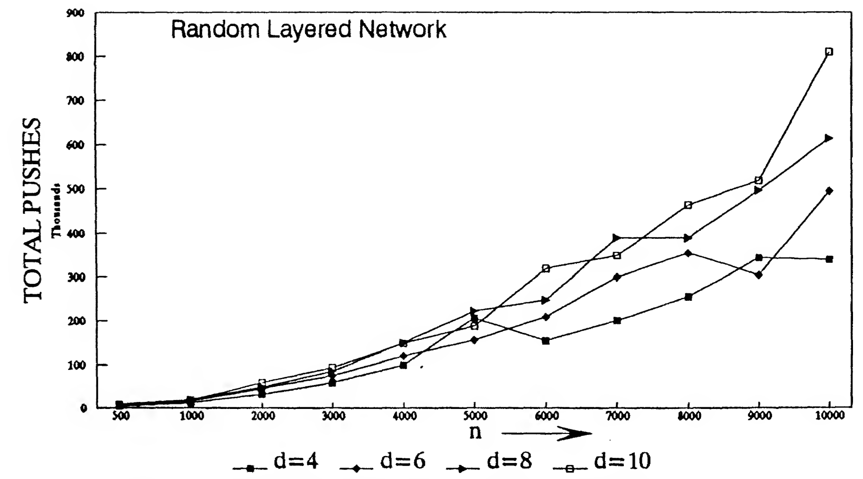


Figure 5.33. Representative operation: total pushes

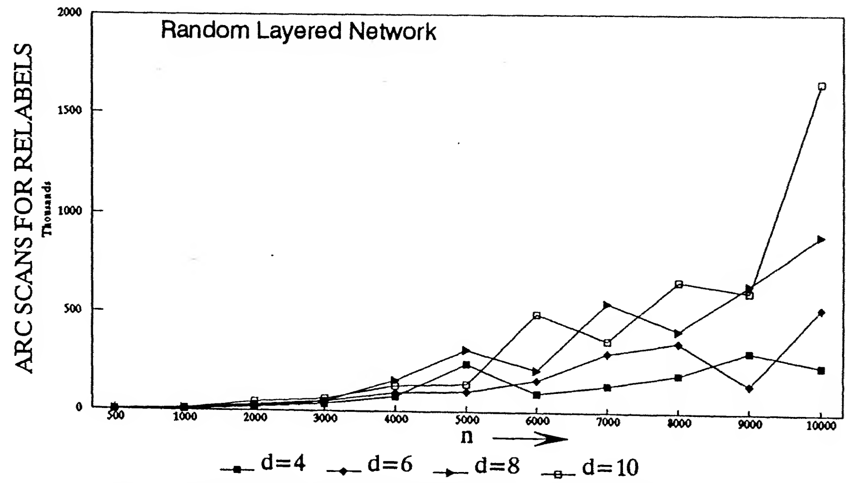


Figure 5.34. Representative operation: arc scans for relabels

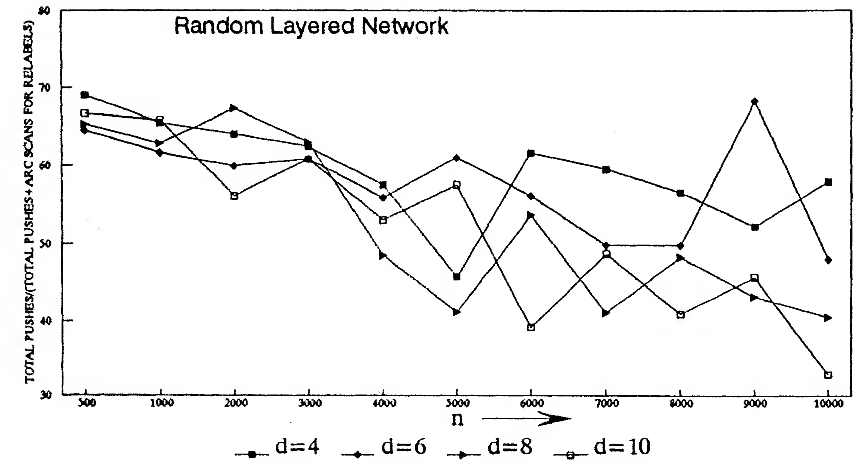


Figure 5.35. Identifying the asymptotic bottleneck operation: ratio of total pushes to total representative opn. counts

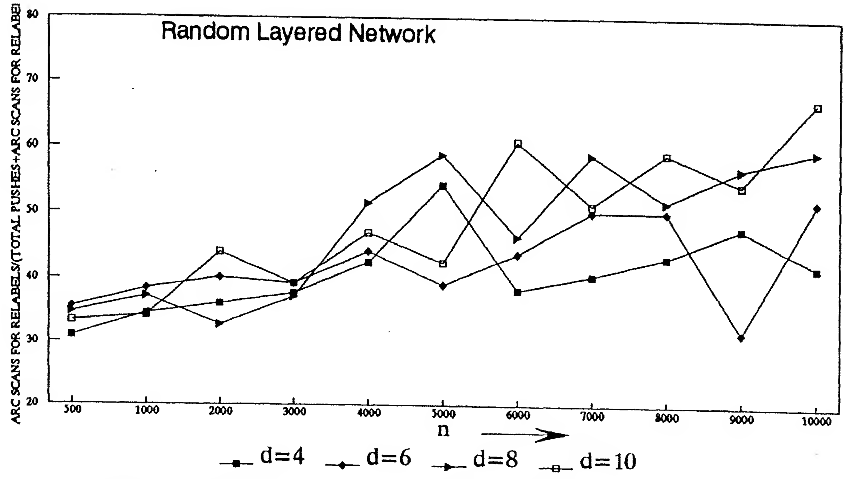


Figure 5.36. Identifying asymptotic bottleneck operation: ratio of arc scans for relabels to total representative opn. counts

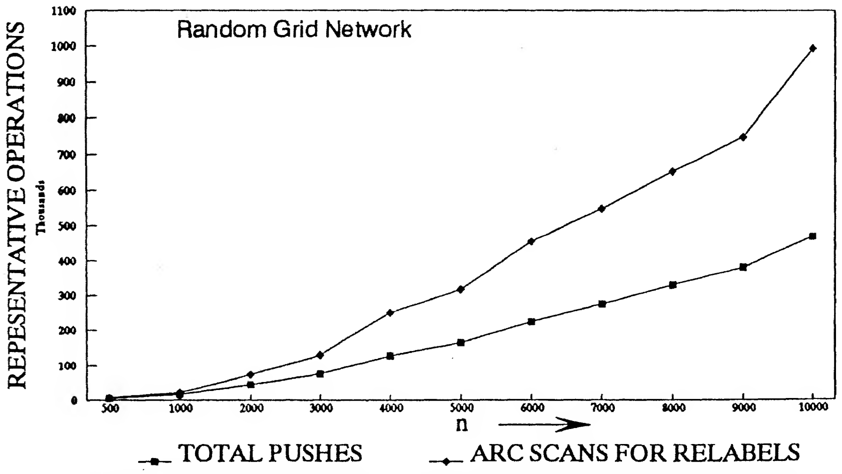


Figure 5.37. Representative operations: total pushes and arc scans for relabels

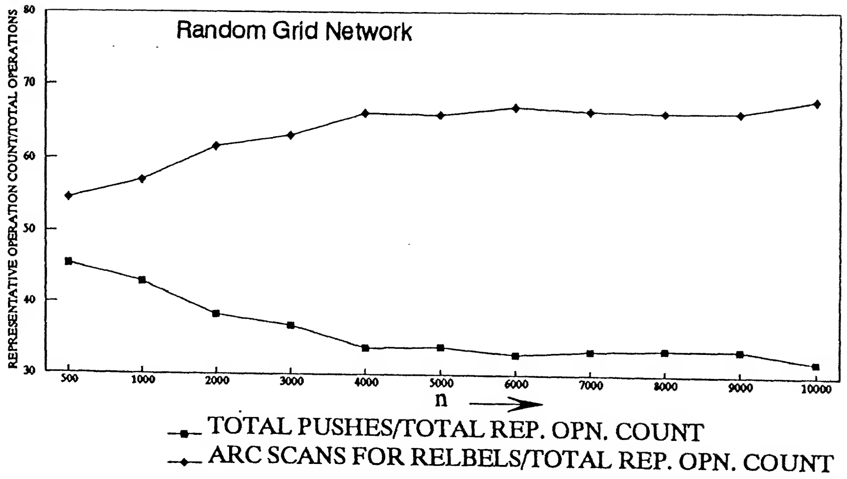


Figure 5.38. Identifying asymptotic bottleneck operation:ratio of representative operation count to total rep. opns. count

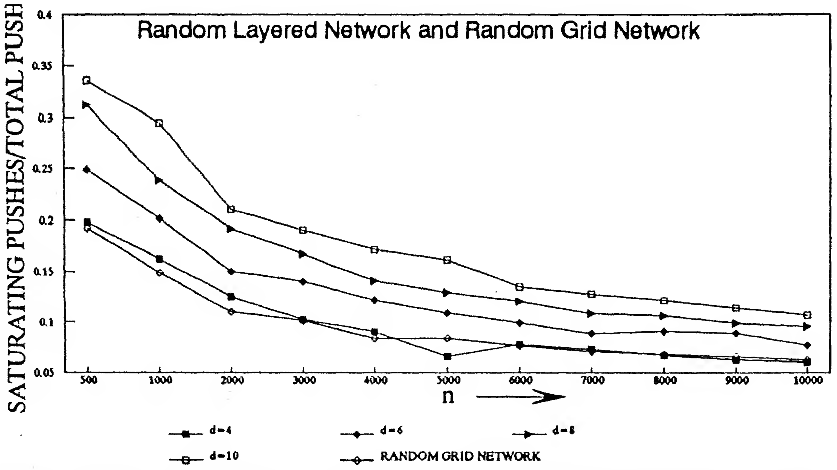


Figure 5.39. Proportion of saturating pushes in total pushes

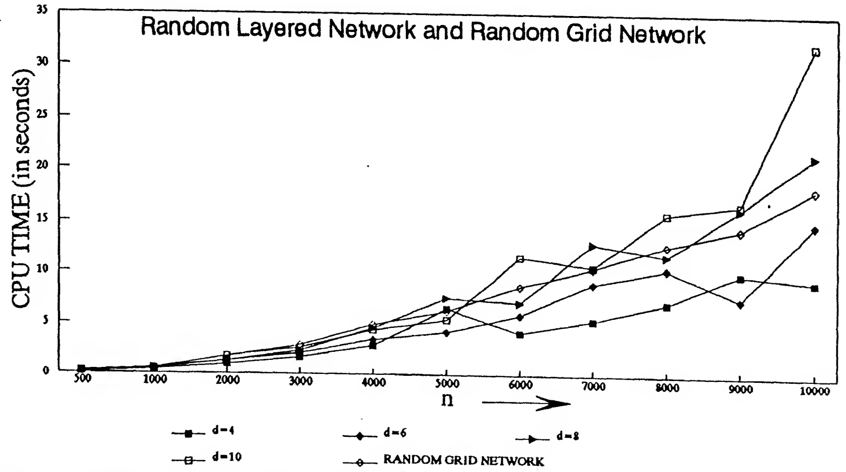


Figure 5.40. CPU time taken

WAVE SCALING ALGORITHM

$V(I) = C_P$ (total pushes) + C_R (arc scans for relabels).

where C_P and C_R are constants. We use multiple regression analysis to estimate C_P and C_R for networks of the same density but diferent sizes.

Network	Density (d)	C_P (10^{-6})	C_R (10^{-6})
Random Layered Network	4	21.7	8.2
	6	21.0	8.6
	8	19.3	10.5
	10	22.0	8.6
Random Grid Network		30.6	3.95

Table 5.3. Regression coefficients in virtual running time estimation for wave scaling algorithm.

We see in Figure 5.41 that the virtual running time does not approximate the CPU time well for networks of size $n \leq 4{,}000$. We see from Figures 5.42 and 5.43 that the pushes are the bottleneck operation for the algorithm for small networks accounting for upto 80%-90% of the virtual running time. Even for networks of size 10,000 nodes the time for pushes takes up 60% of the virtual running time.

Figures 5.44 through 5.47 give an estimate of the growth rate of the various operations as a function of the problem size, n^{γ}. The pushes grow at $n^{1.35}$ to $n^{1.45}$ and the non-saturating pushes grow at $n^{1.4}$.. The arc scans for relabels vary between $n^{1.25}$ and $n^{1.45}$. The relabels vary between n and $n^{1.35}$.

5.8 COMPARISON OF SCALING ALGORITHMS

We compare the three scaling algorithms in this section. All the three algorithms perform around the same number of saturating pushes [see Figures 5.48 and 5.49]. The stack scaling algorithm performs the least number of non-saturating pushes followed by the wave scaling algorithm [see Figures 5.50 and 5.51]. Figures 5.52 and 5.53 show that this is the order for the total number of pushes too. The number of pushes for the excess scaling algorithm is 2 to 3 times more than that for the stack scaling algorithm. We can observe in Figures 5.54 and 5.55 that the saturating pushes by the excess scaling algorithm comprise less than 15% of the total pushes, while they are twice this fraction for the stack scaling algorithm.

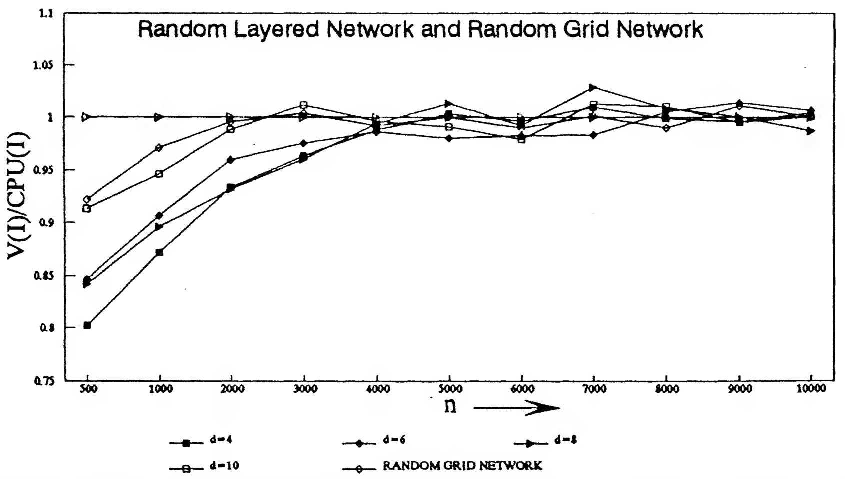


Figure 5.41. Approximation of CPU time by virtual running time

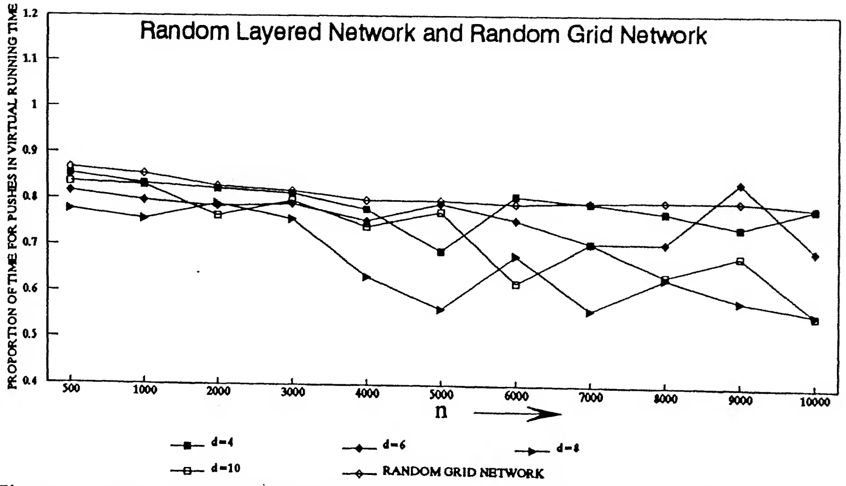


Figure 5.42. Proportion of time for pushes in virtual running time

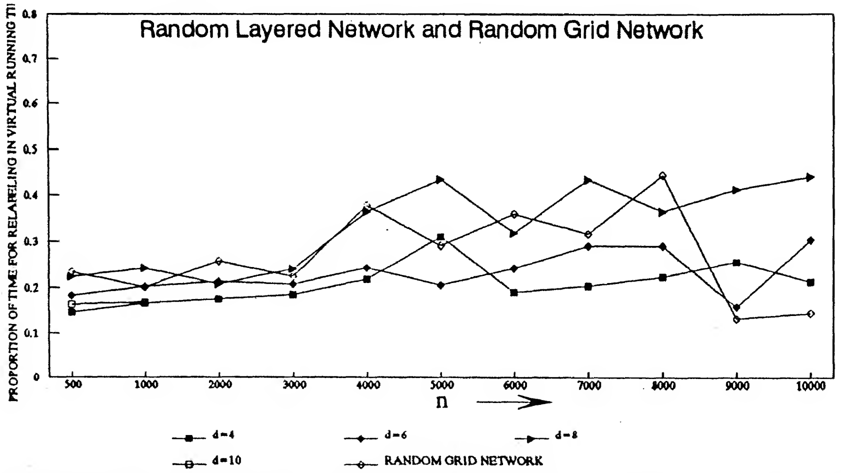


Figure 5.43. Proportion of relabeling time in virtual running time

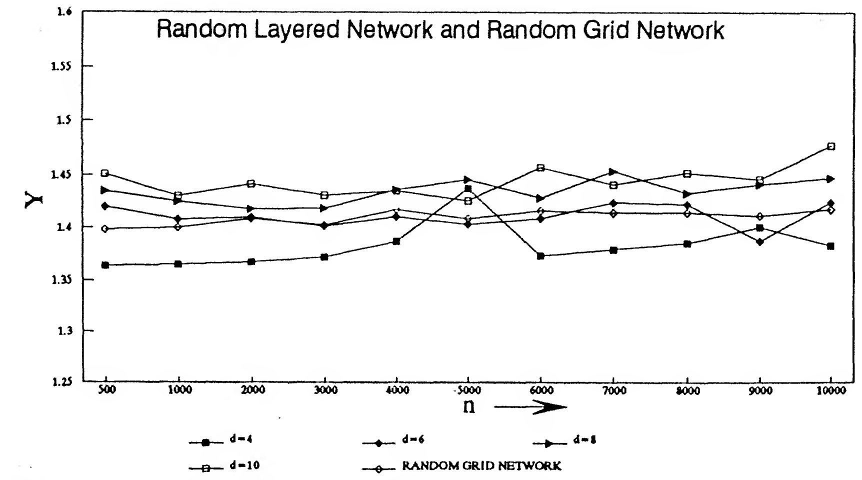


Figure 5.44. Growth rate of total pushes
Y = log(total pushes) / log(n)

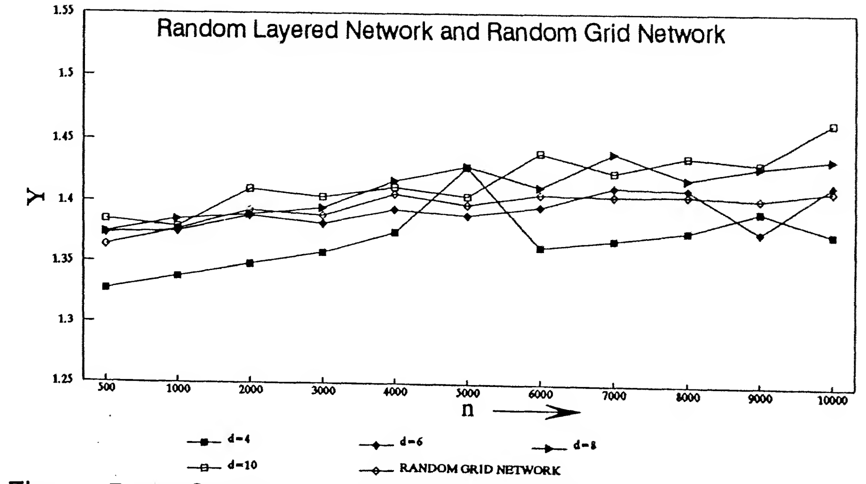


Figure 5.45. Growth rate of non−saturating pushes
Y = log(non−saturating pushes) / log(n)

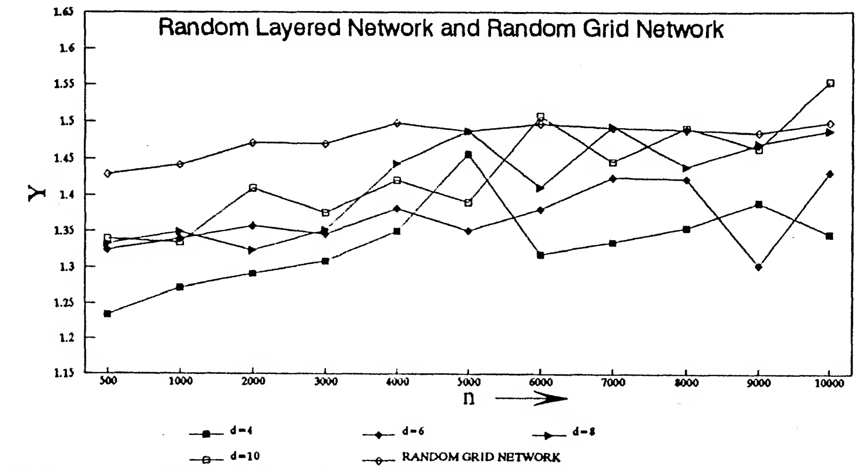


Figure 5.46. Growth rate of arc scans for relabels
Y = log(arc scans for relabels) / log(n)

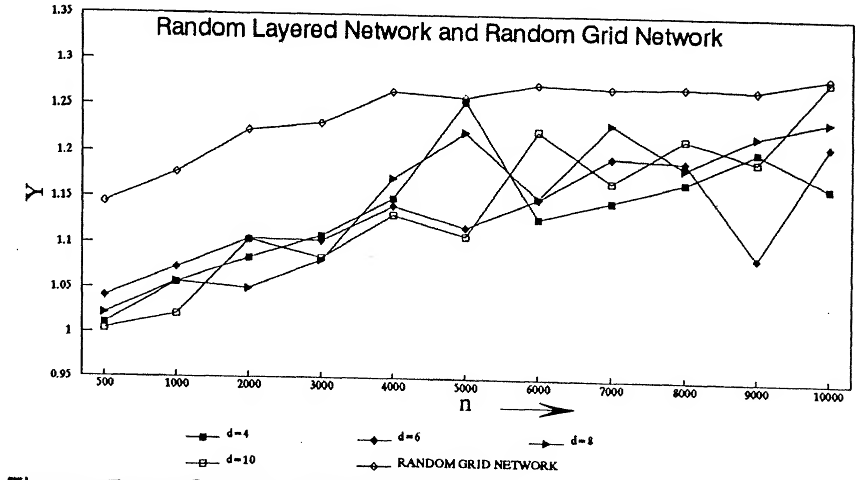


Figure 5.47. Growth rate of relabels
Y = log(# of relabels) / log(n)

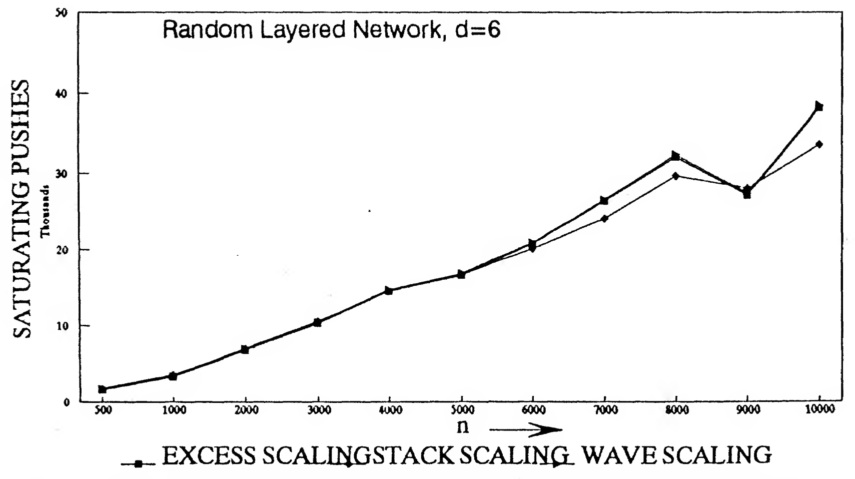


Figure 5.48. Saturating pushes on random layered network

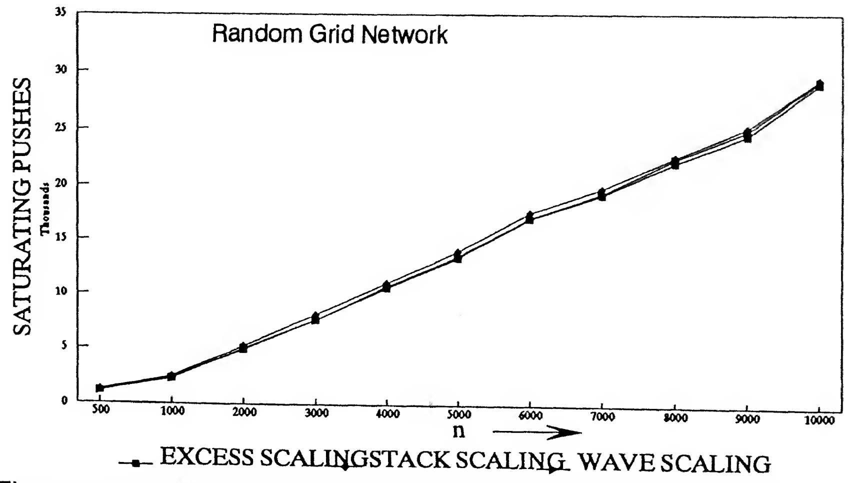


Figure 5.49. Saturating pushes on random grid network

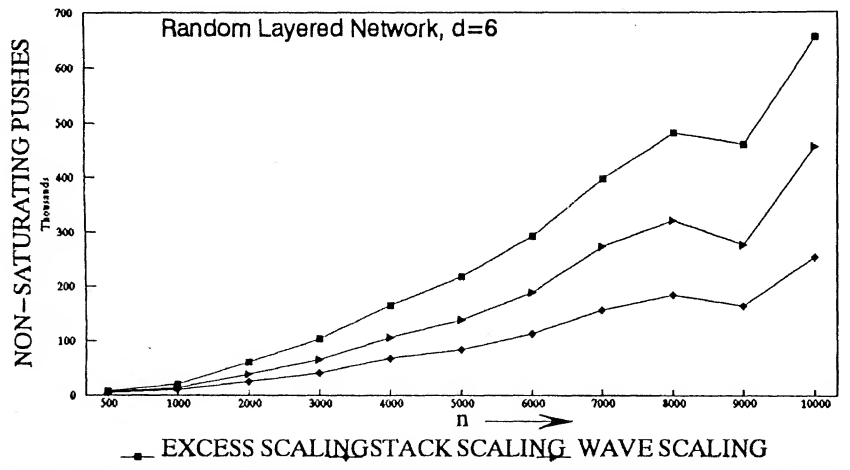


Figure 5.50. Non-saturating pushes on random layered network

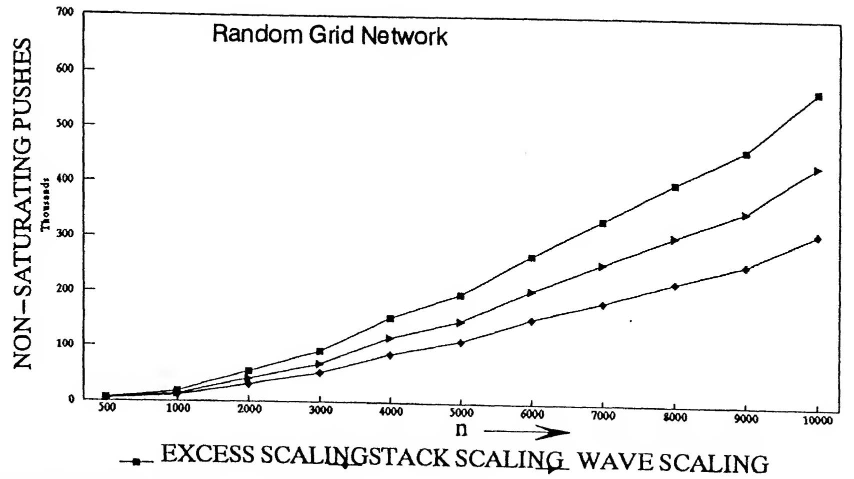


Figure 5.51. Non-saturating pushes on random grid network

COMPARISON OF EXCESS SCALING ALGORITHMS

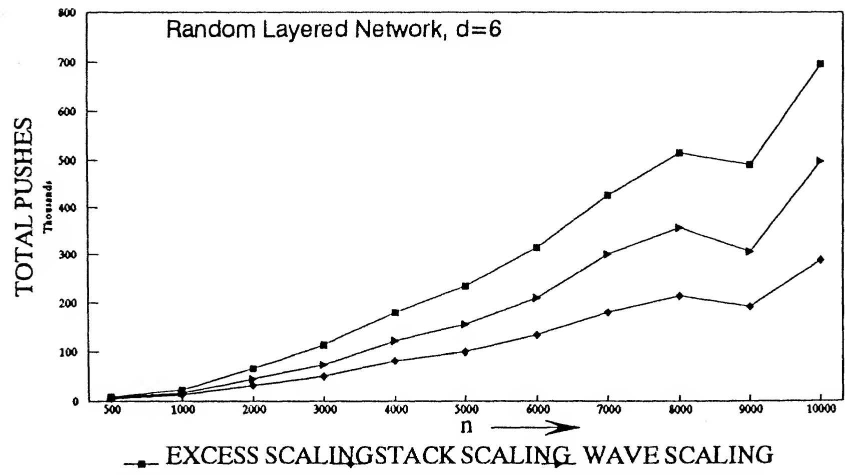


Figure 5.52. Total pushes on random layered network

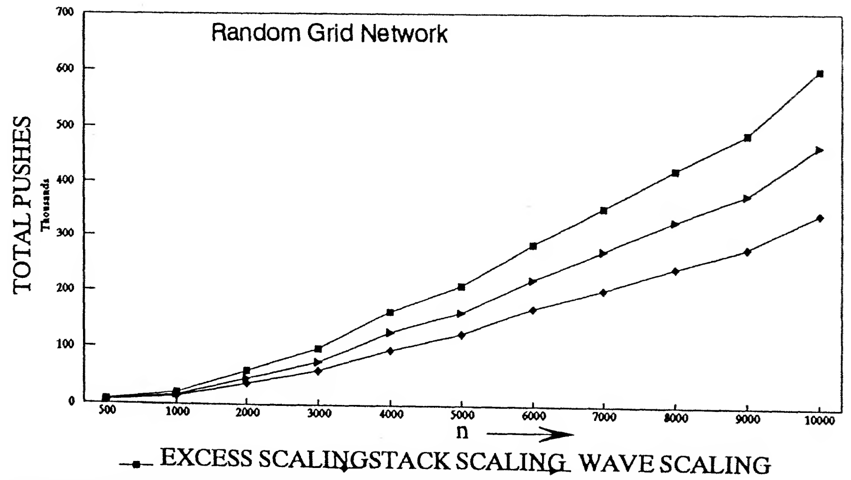


Figure 5.53. Total pushes on random grid network

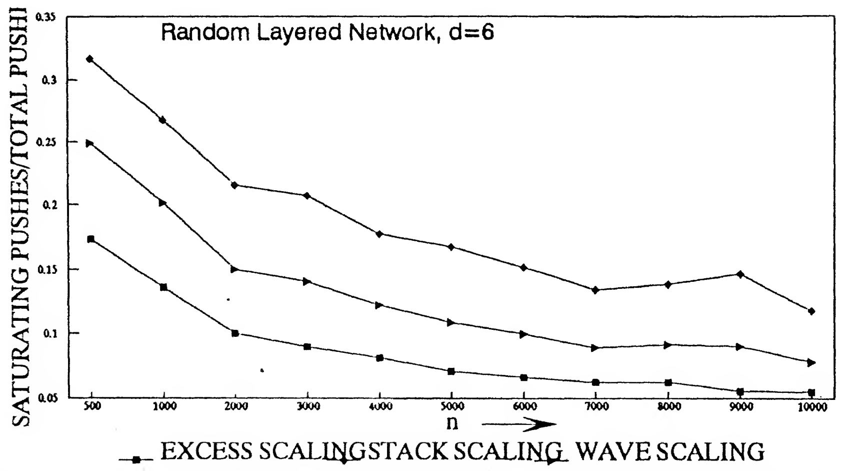


Figure 5.54. Proportion of saturating pushes in total pushes on random layered network

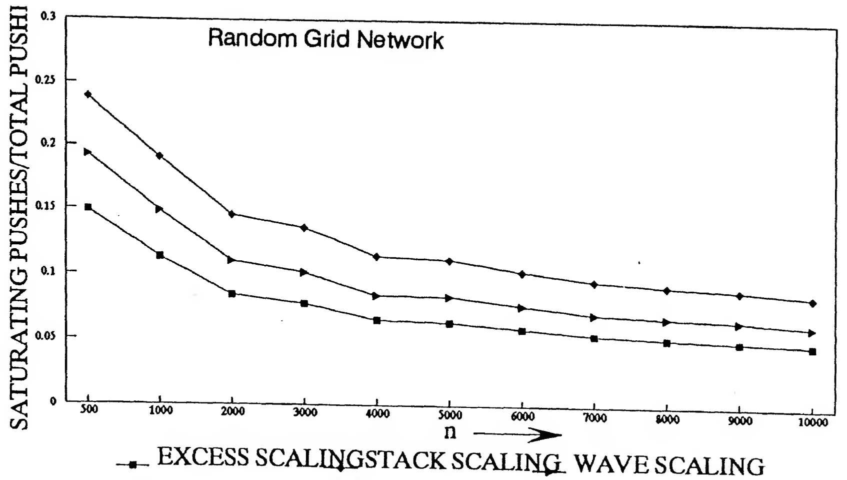


Figure 5.55. Proportion of saturating pushes in total pushes on random grid network

COMPARISON OF EXCESS SCALING ALGORITHMS

The arc scans for relabels are about the same for the excess scaling and the wave scaling algorithms. Stack scaling algorithm performs less than half the arc scans for relabels as the other two for the random layered network and marginally less for the random grid network [see Figures 5.56 and 5.57]

The exces scaling and the wave scaling algorithms take comparable CPU times on the random layered network, while the stack scaling algorithm is about twice faster on the random layered network and is marginally faster on the random grid network than the other two. [see Figures 5.58 and 5.59]

5.9 COMPARISON OF STACK SCALING ALGORITHM WITH EFFICIENT PREFLOW-PUSH ALGORITHMS

In Figures 4.57 through 4.68, we compare the best scaling algorithm with the other good preflow-push algorithms. It is observed that both FIFO and highest label versions of the preflow-push algorithms are better than the stack scaling algorithm. The stack scaling algorithm performs more number of non-saturating pushes, less number of saturating pushes and more total pushes. The stack scaling algorithm has about 20% to 30% staurating pushes in all its pushes, while the FIIFO and highest label version have 30% to 50% saturating pushes of the total pushes they make. The stack scaling algorithm also spends about twice more time on relabeling than the highest label version. The highest label version of the preflow-push algorithm is out 3 to 5 times faster than the stack scaling algorithm and the FIFO version is about 1.5 to 2 times faster than the stack scaling algorithm. The stack scaling algorithm is far superior to Karzanov's algorithm in all respects.

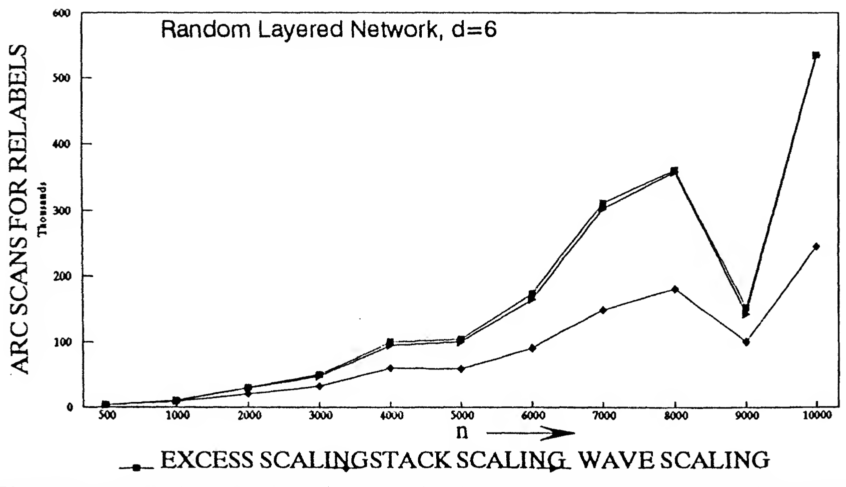


Figure 5.56. Arc scans for relabels on random layered network

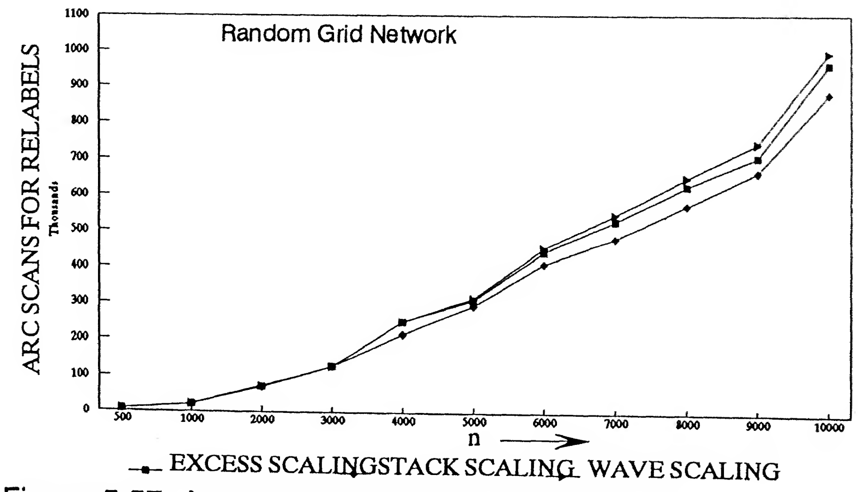


Figure 5.57. Arc scans on random grid network

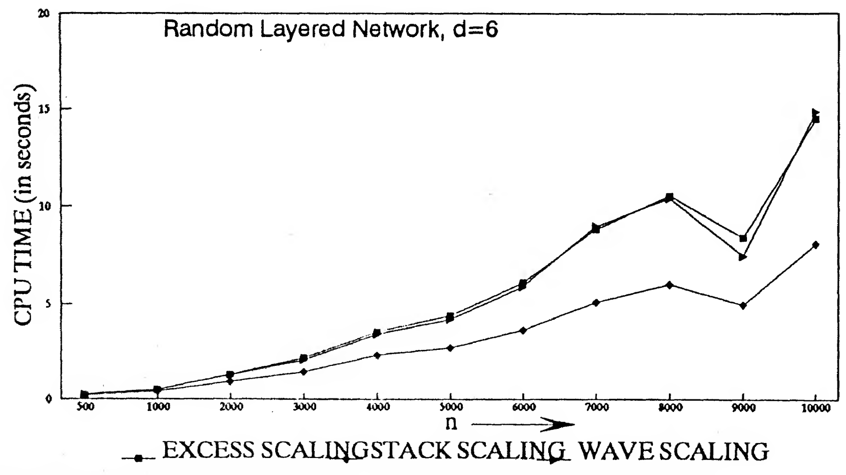


Figure 5.58. CPU time on random layered network

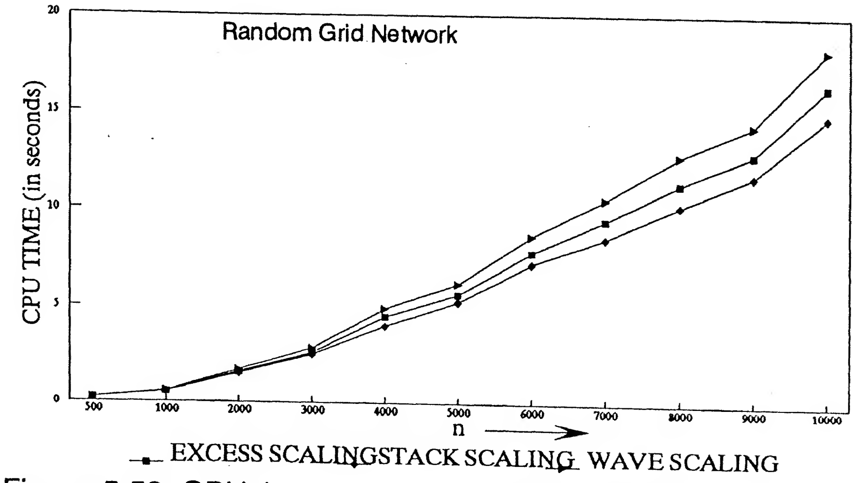


Figure 5.59. CPU time on random grid network

COMPARISON OF EXCESS SCALING ALGORITHMS

CHAPTER 6

CONCLUSIONS

6.1 CONCLUSIONS

We have tested extensively some of the important and computationally efficient augmenting path algorithms and preflow-push algorithms. Some important conclusions of this study are mentioned below.

Augmenting path algorithms

(i) The running time of the Dinic's algorithm and the shortest augmenting path algorithm are comparable, which is consistent with the theoretical proof due to Ahuja and Orlin [1991] that both the algorithms are equivalent in the sense that they will perform exactly the same sequence of augmentations.

(ii) Though in the worst-case, these algorithms perform O(nm) augmentations and take $O(n^2m)$ time, empirically we find that they perform no more than $O(n^{1.3})$ augmentations and their running times are also bounded by $O(n^2)$.

(iii) These algorithms have two bottleneck operations: (i) performing augmentations whose worst-case complexity is $O(n^2m)$; and (ii) relabeling the nodes (or, constructing the layered networks) whose worst-case complexity is O(nm). We observe that empirically the time for relabeling the nodes grows faster than the time for augmentations.

(iv) As the algorithm proceeds, each subsequent augmentation becomes more and more expensive. In fact, for large networks, the algorithm sends 90% of the flow into the sink within 5% of the relabeling of the nodes, and requires 95% of the relabelings to send the remaining flow. A periodic updating of the distance labels to their exact values does not improve the running time and reduces the number of relabels only marginally.

(iv) Incorporating scaling improves the worst-case complexity of these algorithms from $O(n^2m)$ to O(nm logU); but in practice, worsens these algorithms substantially. We find that the capacity scaling algorithm using shortest augmenting paths to send the flow into the sink takes 2 to 4 times the time taken by the shortest augmenting path algorithm. While the capacity scaling algorithm reduces the augmentations made by the shortest augmenting path algorithm by half, the number of relabels increase by about 4 times.

(v) On increasing the scaling factor, we see that the behavior of the capacity scaling algorithm that sends flow along shortest augmenting paths, tends towards the behavior of the shortest augmenting path algorithm.

Preflow-push algorithms

(i) Preflow-push algorithms generally outperform augmenting path algorithms and their relative performance improves as the problem size becomes bigger. For example, the best preflow-push algorithm is superior to Dinic's algorithm by a factor of 30 and superior to Karzanov's algorithm by a factor of 10 for sufficiently large size networks. Thus the recent maximum flow algorithms are an order of magnitude faster than the previous best maximum flow algorithms.

(ii) Among the three implementations of the Goldberg-Tarjan's preflow-push algorithms tested (namely, FIFO, wave, and the highest label pushing), we find that the highest label preflow-push algorithm is the fastest. It is about 2 to 3 times faster than the FIFO version and the wave version is marginally faster than the FIFO version. Among these three algorithms, the highest label preflow-push algorithm achieves the best worst-case complexity and this also translates into an improved empirical performance.

(iii) The preflow-push algorithms have two bottleneck operations: performing non-saturating pushes (which takes $O(n^3)$ or $O(n^2m^{1/2})$ time) and relabeling the nodes (which takes $O(nm)$ time). We find that empirically the time for relabeling the nodes dominates the time for performing the non-saturating pushes. We also find that the non-saturating pushes are only 2 to 4 times more than the saturating pushes.

(iv) Empirically, the running time of the highest-label preflow-push algorithm grows no faster than $O(n^{1.5})$, which is quite attractive.

(v) The time spent by Karzanov's algorithm on constructing layered networks is 10 to 20 times more than the time spent by the Goldberg-Tarjan's preflow-push algorithms on relabeling nodes. Karzanov's algorithm uses about 5 times more the number of total pushes of which only 10% to 20% are saturating as compared to the recent preflow-push algorithms for which the saturating pushes comprise 30%-50% of the total pushes.

(vi) The excess scaling algorithms improve the worst-case complexity of the Goldberg-Tarjan preflow-push algorithms, but this does not translate into an improvement on the empirical front. All the three excess scaling algorithms tested are slower than the above mentioned preflow-push algorithms.

(vii) Of the three scaling algorithms, namely, excess scaling, stack scaling and wave scaling, the empirical performance of the stack scaling algorithm is most attractive. It performs the least number of non-saturating pushes and relabels; and it has the highest proportion of saturating pushes in the total pushes. The stack scaling algorithm is about twice faster than the other two. The wave scaling algorithm is comparable to the excess scaling algorithm.

(viii) Pushing flow from the active node having the highest label is the most attractive implementation of the preflow-push algorithms and of the implementations we tested, pushing flow using the lowest label is the most unattractive. The stack scaling algorithm also pushes flow from the active node having the highest distance label. Excess scaling algorithm incorporates scaling in the lowest label preflow-push algorithm. This reduces the non-saturating pushes by about five times and improves the running time by about three times.

CPU time comparison

Comparing the 11 maximum flow algorithms we tested according to the average CPU time they took to solve the same problem on the random layered network or the random grid network, we can rank them in a decreasing order of speed as follows:

1. Preflow-push algorithm, Highest label version,

2. Preflow-push algorithm, Wave version,

3. Preflow-push algorithm, FIFO version,

4. Stack scaling algorithm,

5. (a) Excess scaling algorithm, and (b) Wave scaling algorithm,

6. Karzanov's algorithm,

7. Preflow-push algorithm, Lowest label version

8. (a) Shortest augmenting path algorithm, and (b) Dinic's algorithm,

9. Capacity scaling algorithm

Table 6.1 presents a summary of the average time taken by these algorithms for different problem instances.

Representative operation count methodology

This is a simple and attractive methodology to test the computational behavior of algorithms. Bottleneck operations for small and large networks can be found and a fair estimate of the growth rate of the bottleneck operations of the algorithm can be made. The virtual running time, for an instance I, is a good approximation of the CPU time.

6.2. SCOPE FOR FUTURE WORK

We have attempted an extensive testing of important maximum flow algorithms using an insightful methodology. A more attentive study of the large amount of meaningful data collected while testing the algorithms and the numerous graphs drawn might lead to many more interesting conclusions regarding maximum flow algorithms. Similar studies can be performed for the other important network flow algorithms. The representative operation counts can be used to characterize network generators.

			SHORTEST				PREFLOW-PUSH ALGORITHMS				EXCESS SCALING ALGORITHMS		
			AUGMENTING	DINIC'S	CAPACITY	KARZANOV'S		HIGHEST		LOWEST	EXCESS	STACK	WAVE
NETWORK	n	DENSITY	PATH	ALGORITHM	SCALING	ALGORITHM	FIFO	LABEL	WAVE	LEVEL	SCALING	SCALING	SCALING
			ALGORITHM		ALGORITHM		VERSION	VERSION	VERSION	VERSION	ALGORITHM	ALGORITHM	ALGORITHM
	200	4	0.06	0.06	0.19	0.03	0.03	0.02	0.03	0.05	0.04	0.04	0.05
RANDOM	500	4	0.21	0.24	0.62	0.14	0.08	0.06	0.08	0.17	0.14	0.13	0.15
LAYERED	1000	4	0.67	0.72	2.05	0.40	0.20	0.15	0.19	0.52	0.36	0.31	0.37
NETWORK	2000	4	2.09	2.19	5.84	1.19	0.49	0.33	0.43	1.60	0.94	0.75	0.93
	3000	4	3.96	4.14	11.52	2.36	0.80	0.50	0.70	3.23	1.59	1.21	1.63
	4000	4	7.27	7.78	20.63	4.81	1.29	0.70	1.12	6.25	2.71	1.93	2.79
	5000	4	13.00	13.80	52.97	9.84	2.67	0.90	2.15	12.78	5.91	3.70	6.50
	6000	4	11.47	12.11	34.52	6.99	1.84	1.05	1.56	9.24	4.05	2.78	4.14
	7000	4	15.45	16.37	41.26	9.67	2.44	1.30	2.04	13.20	5.26	3.61	5.43
	8000	4	19.78	21.01	62.30	13.21	2.98	1.59	2.47	17.98	6.71	4.50	7.13
	9000	4	26.77	28.47	78.22	18.55	4.16	1.77	3.36	25.67	9.08	5.87	10.06
	10000	4	25.64	27.52	68.45	16.43	3.74	1.78	3.16	22.88	8.91	5.79	9.33
	200	6	0.11	0.12	0.32	0.06	0.04	0.03	0.04	0.09	0.06	0.06	0.07
	500	6	0.41	0.45	1.03	0.23	0.11	0.09	0.11	0.32	0.20	0.17	0.21
	1000	6	1.20	1.27	3.12	0.58	0.26	0.19	0.25	0.95	0.49	0.39	0.48
	2000	6	3.58	3.83	8.09	1.76	0.59	0.40	0.54	2.94	1.29	0.90	1.28
	3000	6	6.46	6.86	13.78	3.00	0.92	0.61	0.84	5.22	2.19	1.42	2.03
	4000	6	10.76	11.45	23.65	5.34	1.51	0.87	1.33	9.21	3.54	2.29	3.39
	5000	6	13.78	14.93	26.71	6.45	1.66	1.06	1.44	11.33	4.38	2.68	4.19
	6000	6	19.22	20.30	38.36	9.43	2.20	1.32	1.90	16.54	6.11	3.63	5.92
	7000	6	27.22	29.30	57.09	14.76	3.16	1.56	2.69	25.09	8.86	5.09	9.03
	8000	6	34.63	37.47	76.31	18.64	3.76	1.88	3.18	32.41	10.59	6.06	10.48
	9000	6	29.04	31.14	47.88	12.01	2.93	1.74	2.55	22.76	8.43	4.96	7.51
	10000	6	46.79	49.81	107.92	26.03	5.15	2.30	4.28	44.33	14.58	8.11	14.91
	200	8	0.12	0.13	0.30	0.05	0.04	0.03	0.04	0.09	0.06	0.05	0.07
	500	8	0.51	0.55	1.38	0.22	0.13	0.11	0.12	0.40	0.23	0.19	0.22
	1000	8	1.46	1.59	3.45	0.61	0.29	0.22	0.27	1.14	0.56	0.42	0.53
	2000	8	4.41	4.65	8.06	1.43	0.59	0.47	0.56	3.34	1.43	0.94	1.27
	3000	8	8.63	9.13	16.22	3.05	0.97	0.74	0.91	6.69	2.55	1.58	2.27
	4000	8	15.20	15.93	30.68	6.43	1.73	1.04	1.59	12.89	4.74	2.73	4.55
	5000	8	23.68	25.09	56.43	11.82	3.19	1.46	2.73	21.47	7.27	4.21	7.52
	6000	8	26.66	28.90	45.67	10.94	2.46	1.61	2.17	22.46	7.53	4.22	7.09
	7000	8	41.92	45.42	83.05	20.60	4.22	2.02	3.68	38.63	12.76	6.66	12.98
	8000	8	42.94	46.51	84.73	19.42	3.78	2.12	3.47	37.47	12.00	6.46	11.77
	9000	8	55.32	59.83	108.73	27.47	5.46	2.57	4.65	50.98	16.03	8.44	16.39
	10000	8	68.36	72.52	149.13	32.72	6.79	2.91	5.91	64.73	20.33	10.17	21.55
	200	10	0.14	0.16	0.38	0.06	0.04	0.04	0.04	0.11	0.07	0.06	0.07
	500	10	0.62	0.70	1.56	0.26	0.13	0.11	0.13	0.48	0.25	0.20	0.24
	1000	10	1.71	1.93	3.59	0.58	0.30	0.26	0.29	1.35	0.59	0.44	0.54
	2000	10	6.11	6.42	11.37	2.18	0.76	0.58	0.71	4.82	1.84	1.19	1.69
	3000	10	10.34	11.57	16.75	3.62	1.07	0.84	1.02	8.17	2.94	1.78	2.57
	4000	10	17.93	18.87	33.02	6.12	1.72	1.22	1.58	14.54	4.80	2.74	4.40
	5000	10	23.56	25.85	43.23	7.97	1.94	1.47	1.81	18.79	6.03	3.34	5.39
	6000	10	39.72	41.46	83.89	17.08	4.03	2.01	3.61	35.53	11.28	6.03	11.56
	7000	10	44.22	47.23	75.38	16.41	3.30	2.16	3.02	36.54	11.55	5.88	10.68
	8000	10	59.80	63.52	121.97	25.14	5.12	2.56	4.55	52.03	16.18	8.11	15.81
	9000	10	64.85	69.94	118.98	24.73	4.70	2.72	4.23	54.64	17.49	8.47	16.81
	10000	10	99.24	106.80	220.78	48.50	10.08	3.41	9.21	94.28	31.02	13.65	32.08
	200	5	0.11	0.10	0.52	0.08	0.05	0.04	0.05	0.07	0.06	0.07	0.07
RANDOM	500	5	0.41	0.39	1.71	0.33	0.15	0.11	0.14	0.27	0.21	0.21	0.23
GRID	1000	5	1.25	1.27	4.81	1.02	0.38	0.28	0.36	0.82	0.54	0.54	0.58
NETWORK	2000	5	3.84	3.97	15.17	3.18	1.12	0.76	1.04	2.62	1.54	1.47	1.68
	3000	5	7.80	7.14	33.39	5.54	1.97	1.32	1.80	5.29	2.60	2.49	2.80
	4000	5	15.89	13.82	74.02	12.37	3.14	1.98	2.92	11.67	4.50	4.01	4.93
	5000	5	19.74	18.30	93.14	14.33	4.31	2.89	4.01	13.20	5.69	5.30	6.24
	6000	5	26.80	24.61	110.53	20.05	5.80	3.65	5.40	21.31	7.86	7.29	8.72
	7000	5	33.09	31.64	137.19	25.99	6.74	4.25	6.47	26.35	9.52	8.60	10.58
	8000	5	39.07	40.24	167.13	31.61	8.11	4.88	7.88	30.13	11.36	10.26	12.82
	9000	5	46.81	42.18	202.26	35.85	9.53	5.55	9.23	36.83	12.91	11.81	14.40
	10000	5	67.48	57.37	283.88	51.58	11.43	6.94	11.14	52.41	16.40	14.85	18.24

TABLE 6.1. COMPARISON OF CPU TIME (in seconds) TAKEN BY EACH ALGORITHM

REFERENCES

AHUJA, R.K., AND J.B. ORLIN. 1989. A fast and simple algorithm for the maximum flow problem. *Operations Research* **37**, 748-759.

AHUJA, R.K., J.B. ORLIN AND R.E. TARJAN. 1989. Improved time bounds for the maximum flow problem. *SIAM Journal on Computing* **18**, 939-954.

AHUJA, R.K., and J.B. ORLIN. 1993. Use of representative counts in computational testings of algorithms. To appear in *ORSA Journal of Computing* .

AHUJA, R.K., J.B. ORLIN, AND T.L. MAGNANTI. 1991. Some recent advances in network flows. *SIAM Review* **33**, 175-219.

AHUJA, R.K., T.L. MAGNANTI AND J.B. ORLIN. 1993. Network flows: Theory, Algorithms, and Applications. Prentice Hall, New Jersey.

AHUJA, R.K., AND J.B. ORLIN. 1991. Distance-directed augmenting path algorithms for maximum flow and parametric maximum flow problems. *Naval Research Logistics* **38**, 413-430.

ALON, N. 1990. Generating pseudo-random permutations and maximum flow algorithms. *Information Processing Letters* **35**, 201-204.

CHERIYAN, J., AND S. N. MAHESHWARI. 1989. Analysis of preflow push algorithms for maximum network flow. *SIAM Journal on Computing* **6**, 1057-1086.

CHERIYAN, J., AND T. HAGERUP. 1989. A randomized maximum-flow algorithm. *Proceedings of the 30th IEEE Conference on the Foundations of Computer Science*, 118-123.

CHERIYAN, J., T. HAGERUP AND K. MEHLHORN. 1990. Can a maximum flow be computed in O(nm) time? *Proceeding of the 17th International Colloquium on Automata, Languages and Programming*, 235-248.

CHERKASKY R. V. 1977. Algorithm of construction of maximum flow in networks with complexity $O(V^2E^{1/2})$ operations. *Mathematical Methods of Solution of Economical Problems* 7, 112-115.

CHEUNG, T. 1980. Computational comparison of eight methods for the maximum network flow problem. *ACM Transactions on Mathematical Software* **6**, 1-16.

DANTZIG, G. B., AND D. R. FULKERSON 1956. On the Max-Flow Min-Cut theorem of Networks. In *Linear Inequalities and Related Systems,* edited by H.W. Kuhn and A.W. Tucker, *Annals of Mathematics Study* 38, Princeton University Press, 215-221.

DERIGS, U., AND W. MEIER. 1989. Implementing Goldberg's max-flow algorithm : A computational investigation. *Zeitschrift für Operations Research* **33**, 383-403

DINIC, E.A. 1970. Algorithm for solution of a problem of maximum flow in networks with power estimation, *Soviet Mathematics Doklady* **11**, 1277-1280.

EDMONDS, J., AND R.M. KARP. 1972. Theoretical improvements in algorithmic efficiency for network flow problems. *Journal of ACM* **19**, 248-264.

ELIAS, P., A. FEINSTEIN, AND C.E. SHANNON. 1956. Note on maximum flow through a network. *IRE Transactions on Information Theory* **IT-2**, 117-119.

EVEN, S. 1979. *Graph Algorithms.* Computer Science Press, Maryland.

FORD, L.R. Jr., AND D.R. FULKERSON. 1962. *Flows in Networks.* Princeton University Press, Princeton, NJ.

FORD, L.R., AND D.R. FULKERSON. 1956. Maximal flow through a network. *Canadian Journal of Mathematics* **8**, 399-404.

FULKERSON, D.R., AND G.B. DANTZIG. 1955. Computation of maximum flow in networks. *Naval Research Logistics Quarterly* **2**, 277-283.

GABOW, H.N. 1985. Scaling algorithms for network problems. *Journal of Computer and System Sciences* **31**, 148-168.

GALIL, Z. 1980. An $O(n^{5/3}m^{2/3})$ algorithm for the maximum flow problem. *Acta Informatica* **14**, 221-242.

GALIL, Z. AND A.NAMAAD. 1980. An O(nmlog2n) algorithm for the maximum flow problem. *Journalof Computer and System Sciences* **21**, 203-217.

GLOVER, F., D. KLINGMAN, J. MOTE AND D. WHITMAN. 1979. Comprehensive computer evaluation and enhancement of maximum flow algorithms. *Research Report CCS 356,* Centre of Cybernetic Studies, University of Texas at Austin. See also GLOVER, F., D. KLINGMAN, J. MOTE AND D. WHITMAN. 1980. An extended abstract of an indepth algorithmic and computational study for maximum flow problems. *Discrete Applied Mathematics,* Vol. 2, 251-254.

GOLDBERG, A.V. 1985. A new max-flow algorithm. Technical Report MIT/LCS/TM-291, Laboratory for Computer Science, M.I.T., Cambridge, MA.

GOLDBERG, A.V., AND R.E. TARJAN. 1986. A new approach to the maximum flow problem. *Proceedings of the 18th ACM Symposium on the Theory of Computing,* 136-146. Full paper in *Journal of ACM* 35(1988), 921-940.

GOLDFARB, D. AND M. D. GRIGORIADIS. 1987. A computational comparison of the Dinic and Network Simplex methods for maximum flow. *Technical Report, LCSR-TR-94*. Dept. of Computer Science, Rutgers University, New Brunswick, NJ.

HAMACHER, H.W. 1979. Numerical Investigations on the Maximal Flow Algorithm of Karzanov, *Computing* **22**, 17-29.

IMAI, H. 1983. On the practical efficiency of various maximum flow algorithms, *Journal of the Operations Research Society of Japan* **26**, 61-82.

KARZANOV, A.V. 1974. Determining the maximal flow in a network by the method of preflows. *Soviet Mathematics Doklady* **15**, 434-437.

KLINGMAN, D., A. NAPIER, AND J. STUTZ. 1974. NETGEN: A program for generating large scale capacitated assignment, transportation, and minimum cost flow network problems. *Management Science* **20**, 814-821.

MALHOTRA, V. M., M. P. KUMAR, AND S. N. MAHESHWARI. 1978. An $O(n^3)$ Algorithm for finding maximum flows in networks. *Information Processing Letters* **7**. 277-278.

SHILOACH, Y. 1978. An $O(nI \log^2 I)$ maximum flow algorithm. Technical Report. STAN-CS-78-702. Computer Science Dept., Stanford University, Stanford, CA.

SHILOACH, Y., AND U. VISHKIN. 1982. An $O(n^2 \log n)$ parallel max-flow algorithm. *Journal of Algorithms* **3**, 128-146.

SLEATOR, D.D., AND R.E. TARJAN. 1983. A data structure for dynamic trees, *Journal of Computer and System Sciences* **24**, 362-391.

TARJAN, R.E. 1983. *Data Structures and Network Algorithms.* SIAM, Philadelphia, PA.

TARJAN, R.E. 1984. A simple version of Karzanov's blocking flow algorithm, *Operations Research Letters* **2**, 265-268.